NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

HANDHELD ASSISTANT FOR MILITARY AND POLICE PATROLS

by

Patrick J. Seipel

September 2014

Thesis Co-Advisors: Gurminder Singh
Arijit Das
John Gibson

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 2014	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE HANDHELD ASSISTANT FOR MILITARY AND POLICE PATROLS	**5. FUNDING NUMBERS** W4C12
6. AUTHOR(S) Patrick J. Seipel	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) Naval Postgraduate School Foundation PO Box 8626 Monterey, CA 93943	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB protocol number ___N/A_____.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Military and police patrols are an important component of combat operations, counter insurgency, peacekeeping, disaster relief, and humanitarian assistance missions. These patrols need to access timely, relevant information about events and conditions along their patrol route, both historical and ongoing. In the current practice, this information is gathered manually prior to the commencement of the patrol through the use of historical databases, current event repositories, and by reviewing records that may be relevant to the area to be patrolled. Because it is manual, this process is fraught with numerous problems including high-cost, slow-speed, and low-reliability.

We present an architecture and a prototype system to enhance the effectiveness and security of patrol units, expedite the planning of patrol missions, and reduce the cost of planning. Our system uses commercial off-the-shelf handheld devices and a web-enabled, device-independent software system that enables planning the patrol route and linking related information to that route. Once the patrol starts, the application tracks the unit's current location and provides real-time information and alerts about areas of interest along the route. The command post can track the location of all units and deviations from their planned routes are flagged and the command post is alerted.

14. SUBJECT TERMS Mobile, handheld, geolocation, alerts, device-independent, patrol	**15. NUMBER OF PAGES** 119
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

HANDHELD ASSISTANT FOR MILITARY AND POLICE PATROLS

Patrick J. Seipel
Major, United States Marine Corps
B.S., United States Air Force Academy, 2000

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2014

Author: Patrick J. Seipel

Approved by: Gurminder Singh
Thesis Co-Advisor

Arijit Das
Thesis Co-Advisor

John Gibson
Thesis Co-Advisor

Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Military and police patrols are an important component of combat operations, counter insurgency, peacekeeping, disaster relief, and humanitarian assistance missions. These patrols need to access timely, relevant information about events and conditions along their patrol route, both historical and ongoing. In the current practice, this information is gathered manually prior to the commencement of the patrol through the use of historical databases, current event repositories, and by reviewing records that may be relevant to the area to be patrolled. Because it is manual, this process is fraught with numerous problems including high-cost, slow-speed, and low-reliability.

We present an architecture and a prototype system to enhance the effectiveness and security of patrol units, expedite the planning of patrol missions, and reduce the cost of planning. Our system uses commercial off-the-shelf handheld devices and a web-enabled, device-independent software system that enables planning the patrol route and linking related information to that route. Once the patrol starts, the application tracks the unit's current location and provides real-time information and alerts about areas of interest along the route. The command post can track the location of all units and deviations from their planned routes are flagged and the command post is alerted.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION 1
A. PLANNING FOR PATROLS 2
B. OBJECTIVES 3
C. RELEVANCE TO THE DEPARTMENT OF DEFENSE 4
D. ORGANIZATION 4

II. BACKGROUND 7
A. INTRODUCTION 7
B. CURRENT METHODS OF SUPPORTING PATROLS 7
C. EXISTING SYSTEMS 8
D. BRING YOUR OWN DEVICE 10
E. MOBILE OPERATING SYSTEMS 10
1. Android 11
2. iOS 12
3. Others 12
F. PHONEGAP AND CORDOVA 13
1. History 13
2. Cordova Theory of Operation 13
3. HTML5, JavaScript and CSS3 15
a. Single Page Authoring 16
b. JavaScript Integration with Native Code 17
c. Touch Versus Mouse Events 18
d. Security 19
4. Development Considerations 19
a. Plugin Availability 20
b. User Interface Design 20
G. MAPPING 21
1. Formats 21
a. Proprietary 22
b. Vector Image 22
c. Raster Image 23
d. Global Map Tile Scheme 23
e. Server Generated Tiles 25
f. Locally Generated Tiles 26
2. Mapping Providers 27
3. Map Display Application Programming Interface 28
a. Google Maps API 28
b. Leaflet 29
c. MapBox 30
d. OpenLayers 30
H. ROUTING 30
1. Routing Algorithm 30
2. Routing Service Providers 32

I. CONNECTIVITY CONSIDERATIONS 32
1. Map Cache 33
2. Routing 35
3. Database Replication 35
J. SUMMARY 36

III. ARCHITECTURE 37
A. INTRODUCTION 37
B. SYSTEM ARCHITECTURE OVERVIEW 37
C. EXTERNAL DATABASES 38
D. SUPPORTING COMPONENTS 39
1. Map Server 39
2. Database Server 41
3. Route Server 43
4. Web Server 44
5. Proxy Server 44
E. MOBILE APPLICATION 44
1. User Interface 46
2. Mapping 46
3. Database 46
4. Routing 47
5. Cache 47
F. SUMMARY 47

IV. IMPLEMENTATION 49
A. INTRODUCTION 49
B. APPLICATION DESIGN 49
1. User Interface 49
2. Mapping 50
3. Database 51
4. Routing 51
a. Automated Route Generation 51
b. Manual Route Generation 52
c. Route Checking 52
5. Cache 55
C. APPLICATION FUNCTIONALITY 58
1. Main Interface 58
2. Layers Menu 62
a. Base Maps Tab 62
b. Alert Layers Tab 63
3. Options Menu 64
a. Options Tab 64
b. Create Alerts Tab 65
c. Create / Assign Routes Tab 67
4. Alerts 74
5. Routes 76
D. TESTING 78

1. Devices 79
2. Scenarios 80
a. Usability Testing 81
b. Cache 81
E. SUMMARY 83
V. CONCLUSIONS AND FUTURE WORK 85
A. CONCLUSIONS 85
B. FUTURE WORK 87
1. User Login, Identification, and Authentication 87
2. External Database Integration 88
3. Offline Functionality 88
4. Map Cache 89
5. Dynamic HTML Generation for Alerts 89
SUPPLEMENTARY MATERIAL 91
LIST OF REFERENCES 93
INITIAL DISTRIBUTION LIST 99

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

Figure 1. Cordova directory structure (from Plotz, 2013) 14
Figure 2. Cordova exec interface (from Apache Software Foundation, 2014) ... 18
Figure 3. Vector graphic (after Yug & Cfaerber, 2006) 22
Figure 4. Raster graphic (after Yug & Cfaerber, 2006) 23
Figure 5. Mapping from physical earth to WGS84 projection, (after National Oceanic and Atmospheric Administration & National Aeronautics and Space Administration, 2007; after Stöckli, Vermote, Saleous, Simmon & Herring, 2005) 24
Figure 6. a (Left): Tiling of image at increasing resolution (after Stöckli et al., 2005); b (Right): Tile pyramid (from García et al., 2012) 25
Figure 7. Tile type examples..... 27
Figure 8. OpenStreetMap node, way, and relation example..... 31
Figure 9. Overall system architecture 37
Figure 10. URL addressing scheme 40
Figure 11. Example alert documents 42
Figure 12. Mobile application architecture 45
Figure 13. Subsystems block diagram..... 46
Figure 14. Position is within r distance of P_x..... 53
Figure 15. Position is within r distance of the route, but not P_x..... 54
Figure 16. P_T is on the route segment 54
Figure 17. P_T lies outside the route segment, P2 is the closest point on the segment..... 54
Figure 18. Main interface: browser (left), Android phone (right)..... 59
Figure 19. Waiting for the GPS to become available. 60
Figure 20. Icons for offline, online type unknown, Wi-Fi, 3g, 4g (after Bu, 2011) 60
Figure 21. Recent and non-recent position reports for other units..... 62
Figure 22. Layers Menu icon (Icons-Land, 2014) 62
Figure 23. Base Maps tab, Android phone (left, middle) and iPhone (right) 63
Figure 24. Alert Layers tab 64
Figure 25. Options and Actions Menu icon (from Coelho, 2007) 64
Figure 26. Options tab, browser (left) and Android phone (right)..... 65
Figure 27. Create Alerts tab, browser (left) and iPad 2 (right) 66
Figure 28. Creating a new alert 67
Figure 29. Create / Assign Routes tab..... 68
Figure 30. Four waypoints selected..... 68
Figure 31. Proposed route between waypoints..... 69
Figure 32. Moving a waypoint (circled) 70
Figure 33. Adding a waypoint by clicking on route and dragging (click/drag point circled) 70
Figure 34. Final route after adjustments 71
Figure 35. Adding a manual route 72
Figure 36. Waypoints adjusted and deleted (circled)..... 72

Figure 37. Finalizing a manual route 73
Figure 38. Assigning a route to a unit 74
Figure 39. Example alert information display, browser (left) and Android phone (right) 75
Figure 40. Example alert information display, Android phone (left), iPad 2 (right) 75
Figure 41. Route assignment process 77
Figure 42. A unit that is on the assigned route 78
Figure 43. A unit that is off the assigned route 78
Figure 44. Example of user interface differences: the same button as rendered in four different browsers 79
Figure 45. Dropdown input differences, Android phone (left) and Android tablet (right) 80
Figure 46. Dropdown input differences, iPhone (left), iPad 2 (right) 80
Figure 47. Waterfall chart of download times 82

LIST OF TABLES

Table 1. Smartphone market share, (after Rivera & van der Meulen, 2013) 11
Table 2. Tile count, estimated size, and anticipated download time by zoom level 58
Table 3. Actual download sizes and times for different connection types..... 83

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

API	Application Programming Interface
ASF	Apache Software Foundation
BYOD	bring your own device
C2PC	Command and Control Personal Computer
COP	common operational picture
COTS	commercial off the shelf
CPOF	Command Post of the Future
CPU	central processing unit
CSS3	Cascading Style Sheet version 3
DOM	document object module
FAT	file allocation table
GCCS-J	Global Command and Control System Joint
GIS	Geographic Information System
GPS	Global Positioning System
HA/DR	humanitarian assistance and disaster relief
HTML5	HyperText Markup Language version 5
HTTP	Hypertext Transport Protocol
IDE	integrated development environment
IED	improvised explosive device
JBV	Joint Battlespace Viewer
JS	JavaScript
JSON	Javascript Object Notation
OS	operating system
OSM	OpenStreetMap
OSRM	Open Source Routing Machine
REST	representational state transfer
SDK	software development kit
SVG	scalable vector graphics
TIGR	Tactical Ground Reporting System
URL	uniform resource locator

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

Many thanks to my advisors, Dr. Gurminder Singh, Mr. Arijit Das, and Mr. John Gibson, for the advice and guidance they have given me. Your efforts to enhance my knowledge made the time fly by.

I would also like to acknowledge the support of the Naval Postgraduate School Foundation for sponsoring this research, which was done in collaboration with the Common Operational Research Environment Lab at the Naval Postgraduate School. Without the support of these two organizations, this research would not have been possible.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Military and police patrols are a critical enabler across the entire spectrum of domestic and military operations from community relations to humanitarian assistance and disaster relief (HA/DR) to gang and drug interdiction to counter terrorism and counter insurgency to peace-keeping to full-scale combat operations. Whether during peacetime law enforcement or disaster relief or combat, patrols form the basis of many other types of operations including search, interdiction, security, traffic control checkpoints, and intelligence gathering.

During HA/DR operations patrols serve the important purpose of protecting property, maintaining security, performing search and rescue, and aiding the injured. As discussed by Fuentes and Hunt (2006), after Hurricane Katrina, over 600 state troopers and police officers from New Jersey formed an emergency response team that conducted patrols through the Second, Third, and Sixth Districts in New Orleans. These patrols were primarily focused on search and rescue, but they also assisted with force protection for emergency responders, general police patrols to prevent looting and vandalism, and to assist the Louisiana State Police and the Federal Bureau of Investigation with the collection of intelligence about criminal gangs or groups that might hinder rescue operations. During the response to the 2010 earthquake in Haiti, the United Nations sent troops to patrol the streets to maintain public order and to guard food and other aid deliveries during the relief effort (Lacey, 2010).

The recent conflicts in Iraq and Afghanistan have shown that the future strategic landscape will be one of persistent conflict and that the focus will be less on kinetic operations (i.e., actions involving direct and indirect fires intended to kill the enemy or destroy his resources) and more on counter terrorism, counter insurgency, peacekeeping, and nation building. Success or failure in these types of operational environment is contingent upon maintaining stability and winning the support of the local population (United States Army, 2008). This

necessarily requires close interaction between the forces conducting the operation and the indigenous population. Patrols are invaluable in facilitating this interaction because they enable the unit to gain human intelligence, influence popular opinion, and positively impact the security situation. In July 2010, when Marines fanned out across the Nawa district in Afghanistan's Helmand Province, they used foot patrols to observe and interact with key personalities, observe and document key terrain, and build a positive reputation with the local inhabitants (Flynn, Pottinger, & Batchelor, 2010). These patrols enabled the Marines to understand the social relationships and successfully engage the elders and other powerbrokers in the district, which significantly reduced Taliban influence in the area and led to a 90 percent reduction in Marine and Afghan soldier fatalities in the area of operations (Flynn et al., 2010).

A. PLANNING FOR PATROLS

Military and police patrols have a need for access to timely, relevant information about events and conditions along their patrol route, both historical and ongoing. In many cases, this information is gathered manually prior to the commencement of the patrol from historical databases, current event repositories, and by conducting a review of organizational records that may be relevant to the area to be patrolled. The *Counterinsurgency Patrolling Handbook* (Pennington, 2008) indicates that in order to develop a common operating picture, the patrol leader needs information about the people he will interact with, the history of the area, significant events that have occurred recently, social or religious culture or peculiarities. He needs a current and accurate map that defines the location of roads, bridges, buildings, villages, and key infrastructure. This collection of information becomes a mental framework to enable the patrol leader to evaluate new information and events encountered while on patrol. It should be clear that there is a tremendous amount of information to be collected and reviewed and there could be negative repercussions if the officer misses a critical data point or if that data has not yet been recorded in the sources the officer is reviewing. It is easy to envision a situation where information might be

available from a previous shift, or become available during a shift that, if reviewed by an officer while on patrol, could provide him with the means to disrupt criminal activity, make an arrest, or collect additional information relevant to an open investigation (Bureau of Justice Assistance, 2012).

A commercial-off-the-shelf (COTS) mobile device such as a smartphone or tablet is an ideal candidate to run a software application that would help satisfy these requirements. This handheld assistant could collect and present information from various databases to help the patrol leader develop his mental framework, provide situational awareness alerts during the patrol, and enable the patrol leader to collect and share additional information during the patrol.

B. OBJECTIVES

The goal of this thesis is to develop an architecture and a prototype system to enhance the effectiveness and security of patrol units while at the same time expedite the planning of patrol missions and reduce the cost of planning.

The system will consist of a mobile application that can improve situational awareness for patrols and a web application that can assist the command center with tracking and monitoring the various units under their cognizance. The mobile application will communicate with the command center's web application to share data collected by the patrols and track patrols' progress along their routes. The system will incorporate the necessary functions to assist with gathering information from multiple databases about events, people, and activities along a patrol route, track the progress of the patrol on a map using geolocation, and alert the patrol when they are in the vicinity of those events, people, or activities during the patrol. In addition, as a patrol progresses, the application will allow the patrol to capture new information about persons or items of interest they encounter and share it in real-time with the command center and other patrols via a wireless connection.

C. RELEVANCE TO THE DEPARTMENT OF DEFENSE

While there are numerous databases to track information and several systems that attempt to improve a battlespace commander's situational awareness, there is no application that runs on COTS hardware and provides geolocated situational awareness alerts using data from multiple data sources. A simple COTS-device based tool can improve the planning and execution of patrols. The improved awareness that would result from this application would improve the effectiveness of patrol operations and reduce the risk for the personnel executing the patrol.

D. ORGANIZATION

Chapter I provides a discussion of the need for an application that assists a patrol leader with the gathering of information to prepare for a patrol and providing tracking and alerts to the officer while on patrol. The chapter is made up of two sections: One discusses the idea of a handheld assistant that runs on a mobile device to aid the patrol leader' situational awareness; the other explains the overall objectives of this thesis.

Chapter II provides a description of existing programs and applications related to our research. The discussion includes their strengths and weaknesses and explains how the prototype application fills the gaps that these applications do not. It includes a discussion of the different mobile device operating systems, the Cordova development environment, mapping and routing services, and network connectivity considerations.

Chapter III outlines the architectural design used in developing the prototype system. It explains the overall system design as well as the design of the user application and supporting components.

Chapter IV explains the implementation of the architectural design described in Chapter III. The use of the Cordova development tools and the development and testing of the mobile application are discussed. A walk-through of the prototype application's functionality is provided.

Chapter V provides our conclusions and explores the possible enhancements that could be included in future work. It concludes that developing a device independent mobile application using HTML5, JavaScript, and CSS3 that facilitates improved situational awareness for the leader of a patrol is achievable. The chapter concludes with a discussion of future research that would significantly enhance the capability of the prototype system.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

A. INTRODUCTION

This chapter provides an overview of the concepts that form the background for our research. It discusses current methods for solving the problem, similar existing systems, the dominant mobile device operating systems, and a method for cross-device application development. In addition, it provides background on the creation and display of maps on mobile devices and methods for determining routes.

B. CURRENT METHODS OF SUPPORTING PATROLS

The military has a number of tools and databases that are designed to help commanders develop a common operational picture (COP). These databases are constantly updated by the intelligence portion of the organization, usually the S-2, as additional data is gathered by units in the field. The intelligence analyst will interpret the data to synthesize it into intelligence that will help develop a more accurate picture of the situation on the battlefield (United States Marine Corps, 2003). Battalion-level S-2 shops are adept at collecting human intelligence, signals intelligence, and significant event reports that describe events in the area of operations, like improvised explosive device strikes, ambushes, and insurgent activities (Flynn et al., 2010). The products, developed by the analysts, help commanders determine the impact their units actions will have on the situation.

The current method of preparing for a military patrol requires the patrol leader to be familiar with the current COP. In addition, the S-2 will prepare an intelligence report for the area to be patrolled that outlines key pieces of information, such as known enemy and friendly units, geographic features, priority intelligence requirements, imagery, and map overlays. While much of this information is stored in various automated systems; written reports or oral briefings are still the primary methods of disseminating this information to the

patrol leader prior to the commencement of the patrol (United States Marine Corps, 2003). The patrol leader will review a map of the area and note the location of roads, bridges, buildings, villages, and other key infrastructure as part of his reconnaissance. Marine Corps Warfighting Publication 3-11.3 states:

> For a patrol to succeed, all members must be well trained, briefed, and rehearsed. The patrol leader must have a complete understanding of the mission and a thorough understanding of the enemy and friendly situations. The patrol leader should make a complete reconnaissance of the terrain to be covered (either visual or map), and must issue an order to the patrol, supervise preparations, and conduct rehearsals. (United States Marine Corps, 2000)

The success of the patrol depends on the familiarity the patrol leader has with all of the information he has been provided and his ability to remember and act on it in a potentially stressful situation.

C. EXISTING SYSTEMS

The U.S. military has a myriad of databases and systems designed to store and catalog information that would be relevant to a patrol. Global Command and Control System Joint (GCCS-J) is a set of hardware, software, procedures, and standardized interfaces designed to consolidate intelligence from multiple sources and produce a near real-time picture of the battle-space environment to support joint and multi-national operations (Defense Information Systems Agency, 2014). While some of the data stored by GCCS-J would be useful in the conduct of a patrol, it is focused on providing awareness to commanders directing strategic level operations.

Joint Battlespace Viewer (JBV) is a “software program that maps satellite imagery, maps, and battlefield graphics to the Earth’s surface” (Naval Surface Warfare Center, 2013). It allows the user to display tracks, overlays, icons, routes, and video from other programs, like the Command and Control Personal Computer (C2PC) developed by Northrop Grumman. JBV has the ability to display alerts when a track crosses a boundary. C2PC is a Microsoft Windows

based desktop application that shows COP information, overlays, and unit tracks on a graphical map to enhance situational awareness.

General Dynamics developed the Tactical Ground Reporting System (TIGR), a web-based system used by the U.S. Army that allows soldiers to collect and share tactical-level information between small, mobile, and dismounted units on the battlefield (General Dynamics Inc., 2012). TIGR displays geolocated information on a Google-style map interface and combines data feeds from programs like GCCS-J and Command Post of the Future (CPOF) along with peer-to-peer sharing of collected information from other units using TIGR. It is a situational awareness tool to assist with planning and executing tactical missions.

CPOF is another General Dynamics developed system that incorporates intelligence products, maps, charts, tables, and other planning tools into a single software system to provide battlefield situational awareness to commanders (Paterson, Greenberg, & Green, 2010). CPOF is designed to enable collaboration between multiple units at different levels of command. Like TIGR, CPOF integrates and displays data from other databases in order to support its stated goal of collaborative information sharing.

Each of these systems integrates data from multiple sources in order to aid with planning, increase situational awareness, and provides a context for spatial location of data by displaying routes, tracks, and unit locations using a map. As the most mobile of these systems, TIGR is most similar to our proposed solution to providing situational awareness and mobile alerts to a patrol. Research indicates that TIGR is not designed to run on a mobile platform and does not provide alerts to the user when they are in the vicinity of critical locations, nor does it track the user's location along a pre-defined route (General Dynamics Inc., 2012).

D. BRING YOUR OWN DEVICE

David Willis stated that “Bring-your-own-device strategies are the most radical change to the economics and the culture of client computing in business in decades” (2013, p. 1). Bring-your-own-device (BYOD) policies have become common in the business community and are starting to be recognized by government IT departments as a way to increase employees’ satisfaction by allowing them their choice of device. Gartner, as quoted by Willis, estimates that by 2020 over 45 percent of the business community will fully embrace BYOD and another 40 percent will have policies that support BYOD for some portion of its operations (2013). As mobile devices become more prevalent in society, organizations are beginning to realize that there are cost savings to be had by allowing employees to use their personal mobile devices for business purposes instead of providing a company issued device.

Although Willis discusses many challenges to implementing BYOD, including security, privacy, and limiting migration of data between personal and business domains, one big challenge he spends little time on is fragmentation of the mobile OS environment (2013). As of 2014, there are 10 mobile operating systems available for different devices. When organizations provided mobile devices to their employees, configuration management of devices and applications was inherent in their policy. With BYOD, configuration management becomes much more challenging and expensive. An organization has to determine which operating systems to support. Particularly when the organization has business-specific applications, supporting multiple operating systems can be a code development and configuration management nightmare, as each OS has specific development environments.

E. MOBILE OPERATING SYSTEMS

As of June 2014, there are four major mobile operating systems with significant market penetration: Android, iOS, Blackberry, and Windows Phone

(see Table 1). There is a host of other mobile operating systems including Firefox OS, Sailfish OS, Symbian, Tizen, Bada, and Ubuntu Touch.

Worldwide Smartphone Sales to End Users by Operating System in 3Q13

Operating System	3Q13 Units(Thousands)	3Q13 Market Share (%)
Android	205,022.70	81.9
iOS	30,330.00	12.1
Microsoft	8,912.30	3.6
BlackBerry	4,400.70	1.8
Bada	633.3	0.3
Symbian	457.5	0.2
Others	475.2	0.2
Total	**250,231.70**	**100**

Table 1. Smartphone market share, (after Rivera & van der Meulen, 2013)

1. Android

The Android operating system holds the overwhelming share of the mobile device market, with 81.9 percent of the market share as of third quarter 2013 (Rivera & van der Meulen, 2013). Android was developed by Google based on the Linux kernel and is “an open-source software stack created for a wide array of devices with different form factors” (Android Open Source Project, 2014b). The Android Software Development Kit (SDK) includes all of the Android APIs packaged with an Eclipse Java based Integrated Development Environment (IDE). Android applications are normally written in Java and compiled to Java byte-code before being packaged and loaded onto a device. The Android Native Development Kit does allow libraries and applications written in other languages to be compiled to code native to processor chipsets, such as x86, ARM, or MIPS, but this is discouraged since the Android Java Virtual Machine is highly optimized, writing native code applications is highly complex, and only CPU-bound applications gain any significant speed advantage (Android Open Source Project, 2014a). The Android SDK includes an emulator to enable application developers to test their applications on a variety of mobile device configurations without having to acquire actual devices.

2. iOS

Apple's iOS is the second most prevalent mobile operating system with 12.1 percent of the market share as of third quarter 2013 (Rivera & van der Meulen, 2013). iOS is a closed-source operating system and is proprietary to Apple Inc. Application development for iOS requires the developer to have a Macintosh computer running OS X, to use the current Xcode IDE, and have the iOS SDK installed on the computer (Apple Inc., 2013).

iOS applications must be written in Objective-C and use the Cocoa Touch user interface library. The Xcode IDE includes an emulator and a developer can deploy its application to an emulated iPhone or iPad emulator for testing.

Apple devices require all applications to be digitally signed by an approved developer certificate, so to deploy an application to a real device the developer must purchase an Apple Developer License in order to obtain a code-signing certificate (Apple Inc., 2014b).

3. Others

Windows Phone and Blackberry OS each have less than five percent of the smartphone market, followed by several even less popular mobile operating systems, including Tizen, Bada, Symbian, Firefox OS, and Ubuntu Touch. Windows Phone, Blackberry OS, Bada, and Symbian are all closed-source, proprietary operating systems. Tizen, Firefox OS, and Ubuntu Touch are Linux-based, open-source platforms for mobile devices. All of these mobile operating systems come with their own SDK for developing applications. Blackberry OS, Bada, Tizen, Sailfish OS, Windows Phone, and Ubuntu Touch use C, C++, or C# as a development language. Notably, Firefox OS applications are written entirely in HTML5, CSS, and JavaScript with enhanced access to the device's hardware and services provided by the Firefox OS API (Mozilla Foundation, 2014).

F. PHONEGAP AND CORDOVA

PhoneGap and Cordova are mobile application development frameworks that enable a developer to write the code for an application in HTML5, CSS3, and JavaScript and then deploy it to a variety of mobile devices without having to re-code the application into the native programming language for that platform.

1. History

PhoneGap was originally developed by Nitobi Software, which was acquired by Adobe Systems in 2011 (Adobe Systems Inc., 2011). Concurrent with the acquisition, Adobe contributed the PhoneGap source code to the Apache Software Foundation (ASF) in order to facilitate continuing improvement by the open source community (Adobe Systems Inc., 2011). In order for Adobe to maintain a clear trademark and meet ASF's license for open source software, the open source version was renamed Apache Cordova. PhoneGap is currently a downstream distribution of the Apache Cordova project, with Adobe having license under the PhoneGap trademark to add additional and proprietary value-added services, such as the Adobe PhoneGap Build online compilation platform and integration with its other web-authoring tools (Leroux, 2012). PhoneGap and Cordova both provide the same essential functionality. Therefore, for consistency, the development framework for this thesis will be referred to as Cordova.

2. Cordova Theory of Operation

The Cordova framework consists of a set of command line tools and software libraries. When a new Cordova project is created, Cordova creates a specific set of directories, as depicted in Figure 1, each with a specific function. The developer places his or her application's code in the "www" directory. Cordova manages the other directories during the build process, moving and replacing files as necessary.

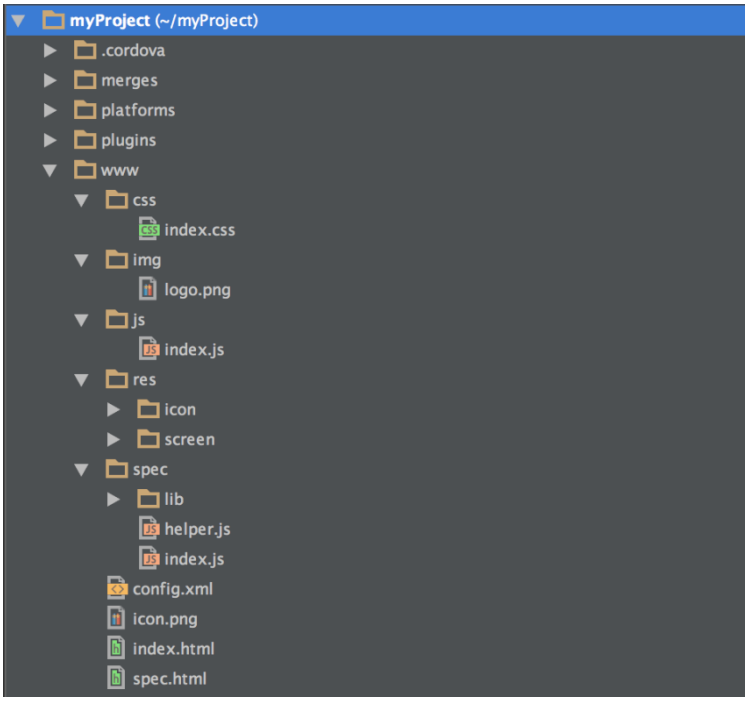

Figure 1. Cordova directory structure (from Plotz, 2013)

In order for Cordova to compile an application for a specific mobile device operating system, the development system must have the Software Development Kit (SDK) loaded for that particular mobile operating system (Apache Software Foundation, 2014). During the build process, Cordova scripts the running of the SDK tools to compile a native code application that consists of a WebView and a foreign function interface that enables the developer's JavaScript code to access native functions, referred to by Cordova as plugins (Apache Software Foundation, 2014). These plugins are native code functions that enable access to mobile device hardware or functionality that is not otherwise exposed by the WebView to the JavaScript code. The Cordova build process packages the contents of the "www" directory into application resources, compiles the appropriate native code for the target mobile operating system, and then assembles it all together into the appropriate type of package to be loaded onto the mobile device.

As described in the Cordova user documentation (Apache Software Foundation, 2014), when a Cordova application is run on a mobile device, the

Cordova native code instantiates and configures a full-screen WebView and then directs it to load the developer's index.html file located in the application's resources. While the WebView is loading and rendering the HTML, Cordova's native code continues in a separate thread to load any required plugins and establish the foreign function interface. This asynchronous loading process makes the application appear to run more quickly, but raises the possibility that the WebView may finish loading and begin running the developer's JavaScript code before Cordova has finished its loading and setup process. To prevent the JavaScript code from attempting to access a native function that has not been loaded yet, the developer is required to wait for an *OnDeviceReady* event to be fired by Cordova before accessing any of Cordova's native functions. When Cordova has finished its setup, it passes the *OnDeviceReady* event to the WebView.

3. HTML5, JavaScript and CSS3

Cordova's use of HTML, JavaScript and CSS enables a developer to use the same skillsets for developing a mobile application as he would to develop webpage. Most of the same concepts discussed by Frain (2012) that are applicable to responsive web page design are applicable to the design of a Cordova mobile application. All of the modern mobile WebView-enabled browsers support HTML5 and CSS3. HTML5 and CSS3 bring additional capabilities that enable a Cordova application to have fluid layouts that adapt to different viewport sizes; support CSS3 typography, transformations, transitions, animations, and other visual effects; and access a number of non-traditional APIs without writing native code, including local file access, geolocation, media, web storage, and others. WebView supports any HTML, CSS, and JavaScript that will run on that device's mobile browser version. There are a number of additional considerations that must be addressed for a Cordova-based application, however, including single page authoring, integration with the mobile device hardware, user interface design, and managing the differences between touch and click events.

a. Single Page Authoring

While the Cordova framework does provide the ability to load a different page, this will disrupt the user experience when the WebView loads and renders the page (Apache Software Foundation, 2014). In addition, loading a new HTML page clears the JavaScript stack; if the application needs to persist data across the reload, the application will have to use cookies, store the data in the window object, or use local storage to store and then re-load the data. All of these approaches will impact the responsiveness of the application. It is for these reasons that most Cordova applications use a method known as single page authoring.

Single page authoring is a method of web page authoring where the browser retrieves only a single HTML web page at the beginning of the session. All changes to the page after that point are done dynamically, using JavaScript to add and remove elements from the document object model (DOM). By adding and removing elements from the DOM, the user interface can be manipulated using all of the elements familiar to the user including menus, buttons, popup dialog boxes, etc. without requiring the browser to perform another full-render on the page. This causes the page to appear much more responsive and does not require workarounds to get JavaScript variables to persist across user interface changes.

Single page authoring for mobile devices does require additional consideration for DOM complexity. Due to resource limitations on mobile devices, the child-depth limit for DOM objects imposed by either the HTML parse engine or device memory might become an issue if the developer chooses to hide objects as opposed to completely removing them from the DOM. One method for speeding up the initial page load and render is to use a framework that only loads the initial page view. Additional content can be dynamically loaded from additional HTML files just prior to when it is needed.

b. JavaScript Integration with Native Code

HTML5 and CSS3 provide the ability to develop spectacular and responsive user interfaces. For an application that needs to do nothing more than display a nice user interface and handle some user inputs, HTML5 and its related API's provide more than enough capability. For an application that needs more direct control over the hardware the WebView sandbox can be limiting. HTML5 provides some limited access to the mobile device's camera, GPS, database storage, and file system. These API's are limited both in the application's ability to control them, and by whether a particular mobile browser supports them. Mobile devices today have additional sensors and capabilities including Bluetooth, Near Field Communication, Wi-Fi Direct, accelerometers, light sensors, proximity sensors, etc. that have no HTML5 or JavaScript API to enable access to them via the WebView.

The WebView component provides the ability to access native code functions from the JavaScript code running in the WebView. Cordova exposes this functionality using a standardized plugin framework. Plugins must be developed in the native language for the device to be targeted and include a native component that accesses the device hardware and a JavaScript interface that invokes the Cordova exec function to access the native interface (Apache Software Foundation, 2014). The Cordova exec function takes five arguments: a success callback, an error callback, a service name, an action name, and an array of arguments (see Figure 2). The service name is the name of the native class, and the action name is the method of the native class that should be called. The arguments are passed to the action method. This interface enables a Cordova application to use plugins to access device hardware or provide capability that is not exposed through HTML5 or its related APIs.

```
cordova.exec(function(success) {},
        function(error) {},
        "service,"
        "action,"
        ["firstArgument," "secondArgument," …, lastArgument]);
```

Figure 2. Cordova exec interface (from Apache Software Foundation, 2014)

c. Touch Versus Mouse Events

There is a significant difference in the design requirements for an application where user interaction is accomplished via touch versus an application where the user has a mouse and keyboard. Most applications designed for display on a desktop browser anticipate that the user is interacting with the page using a keyboard and mouse, although touch is rapidly moving into the desktop space, and touch events should be considered. On a mobile phone or tablet, the user is most likely to be using touch, multi-touch, or a stylus to interact with the application.

While Fitts's law (1992) applies to both mouse and touch interaction, a mouse cursor is relatively more accurate than a finger, so application interfaces designed for touch interaction must have larger controls and more control spacing (Forlines, Wiggdor, Shen & Balakrishnan, 2007). Newer desktop browsers support both mouse and touch events and mobile browsers will simulate mouse events based on touch events if the touch events are not handled. Unfortunately, each browser handles the translation a little differently (Koch, 2014), so the user experience may not be uniform on each platform. In particular, many browsers have a built-in 300 millisecond delay before turning a touch event into a mouse event to determine if the user is performing a double-tap; this delay can cause the interface to feel sluggish if it is not overridden (Wilson & Kinlan, 2013).

d. Security

The use of Cordova to develop mobile applications introduces a number of security challenges including cross-origin policy restrictions and breaching of the sandbox model. By design, web browsers do not allow a script from one domain to make a request for content or data from another domain. This presents a problem for a WebView application that wants to retrieve and manipulate or display data from a server since its domain is defined as either “file://” or “http://localhost.” Mobile operating systems that support the WebView concept allow the application to provide a whitelist of acceptable domains from which to retrieve data. The application may still be limited in what it can do with data retrieved from a server that does not support Cross-Origin Resource Sharing headers, however. For example, an application using the JavaScript XMLHttpRequest function to retrieve an image from a server that does not set the Access-Control-Allow-Origin response header will be limited by the WebView in what it can do with that image to prevent malicious cross-site scripting or code injection.

The WebView functionality that enables JavaScript code running in the WebView to interface with native code breaches the browser sandbox model by design. Unfortunately, as discussed by Luo, Hao, Du, Wang, and Yin (2011) this interface could enable a malicious application on the device to manipulate the JavaScript being run in the WebView or allow a malicious script that is loaded by the WebView, perhaps due to the user clicking a link, to access the native code interfaces exposed to the WebView.

4. Development Considerations

There are some development considerations that must be evaluated before deciding whether to use Cordova to develop a mobile application. These considerations are the availability of plugins and the design of the user interface.

a. Plugin Availability

The Cordova API provides plugins to access many of the most popular hardware interfaces, operating system components, and system events (Apache Software Foundation, 2014). Hardware interfaces include the device's battery status, the accelerometers, the compass, the camera, vibration, network status, and the GPS. Operating system functions that can be accessed include the file system, the user's contacts list, globalization functions, the operating system's email client, and an in-application browser. System event plugins include application pause and resume; changes in online status; and volume, home, and back button presses. Any functionality that a developer wants to add to an application that is not covered by the Cordova provided plugins requires the developer to create a plugin to access that capability. In particular, Cordova plugin support is lacking for several common communication interfaces on mobile devices, including Bluetooth, Wi-Fi Direct, Near Field Communication, and socket IO.

b. User Interface Design

The Cordova API provides the ability to use native dialog boxes on each of its platforms to assist with making the user interface feel more like a native application for that platform than a web-based application (Apache Software Foundation, 2014). This might be important if the application is to be submitted to a commercial app store. For example, Apple requires all applications to conform to its user interface guidelines, and applications that fail to do so are rejected (Apple Inc., 2014a). Another option for developers is to completely manage the entire user interface, developing all dialog boxes, popups, etc. using HTML and CSS. While this approach will not mimic the platform's native user interface, it will enable the developer to create a consistent user interface across all of the devices. The advantage to this approach is that once a user is familiar with the interface, transitioning to a new device will be seamless, as the user interface will be exactly the same.

G. MAPPING

A map that is displayed to the user in an application that allows them to zoom and pan is generally referred to as a slippy map. The OpenStreetMap Project defines a slippy map as a web-browser-based map interface that enables a user to zoom and pan a map by grabbing the map with the mouse and sliding the map image in any direction. The web browser dynamically loads the new portion of the map display without reloading the page, making for a seamless user experience ("Slippy map," 2014). The map image is built out of many smaller images referred to as tiles. These map tiles may be in a number of different formats and may be stored locally, produced on a server, or generated locally as needed. Each tile is referenced by its location in relationship to a grid and its zoom level. This makes it possible to take given latitude, longitude, and zoom level, and determine which map tile needs to be displayed to show that location.

1. Formats

The actual map data that describes the geographic features can be stored in many formats including Keyhole Markup Language, GeoJSON, ArcGIS, PostGIS, and many others. These formats are all standards for encoding geographic information into a standardized file format. All contain information about features on the surface of the earth along with their spatial location. These Geographic Information System (GIS) files are used to generate the actual map tiles that are displayed for the user.

Map tiles can be generated in three basic formats: proprietary, vector-based images, and raster-based images. They can be generated locally on the device that will display them or generated on a server and delivered to the device over the network.

a. Proprietary

Proprietary formats are used by commercial GPS providers, such as Garmin, Lowrance, and Navikey. Based on reverse engineering and open source information, these formats combine vector display information for the map features, elevation data, and routing information to support GPS navigation ("OSM map," 2013).

b. Vector Image

The most efficient way to store map tile information is using vector-based images. A vector image format describes the image using points, lines, angles, curves, and polygons, along with color information. This format is advantageous because it has a small file size and, as depicted in Figure 3, vector images do not lose information as they are scaled up or down. The disadvantage to vector images is that they can only store images based on shapes, and are not useful for displaying photograph style images. In addition, they must be rendered each time they are displayed, moved or scaled, so a large or complex image may require significant processing power or incur a delay in the display to the user.

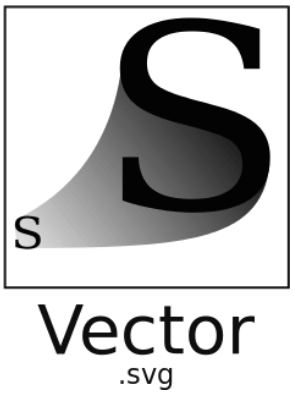

Figure 3. Vector graphic (after Yug & Cfaerber, 2006)

Vector images are ideal for representing man-made geospatial features such as roads, buildings, etc. because all of the points in the file can be geographically referenced. Zooming the image then becomes just a matter of

adjusting the scale of the image and redrawing it; all of the associated shapes will scale appropriately.

c. Raster Image

Raster images, also known as bitmaps, have a grid data structure that stores the color information required to display each pixel of the image. Raster images are most advantageous when there is a need to display photo-realistic images with significant color gradients. The file size of raster images is generally larger than a similar vector image, and while raster images can be compressed, this introduces another processing step. Raster images also do not scale well to higher resolutions. As seen in Figure 4, raster images suffer from pixilation when scaled, or zoomed-in beyond the number of pixels defined by the raster image.

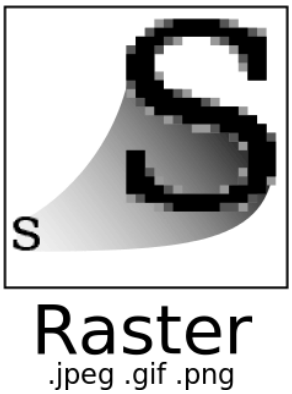

Figure 4. Raster graphic (after Yug & Cfaerber, 2006)

Raster images for satellite or aerial imagery can be geolocated by providing a bounding rectangle for the image that describes the area of the earth's surface covered by the image, or by providing the geographic coordinate of one of the image's corners and specifying the distance covered by a pixel in each direction (Sample & Ioup, 2010).

d. Global Map Tile Scheme

Tile-based mapping systems require that a depiction of the earth's surface be decomposed into a logical set of discrete tiles that can be addressed via a

coordinate system. This requires taking the roughly ellipsoid earth and mapping it to a flat surface using a map projection, as seen in Figure 5. The most popular projection used in online maps is the World Geodetic System 84 Pseudo-Mercator projection, which was adopted and popularized by Google (Google Inc., 2014).

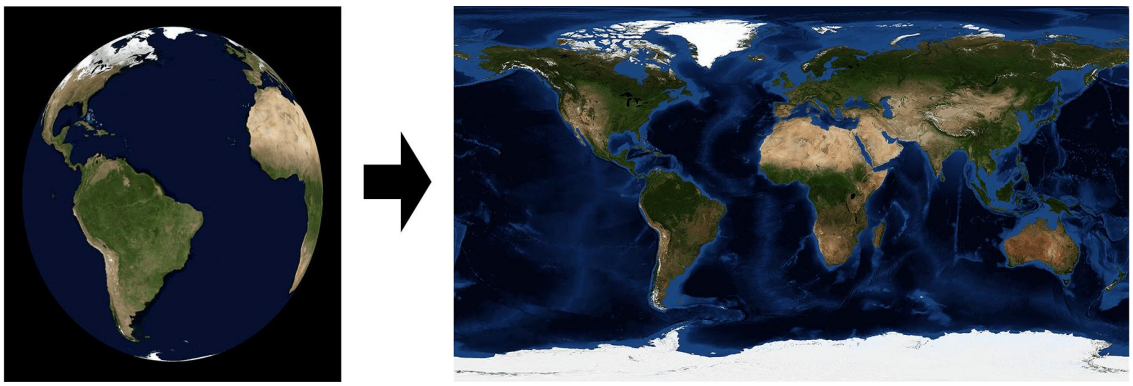

Figure 5. Mapping from physical earth to WGS84 projection, (after National Oceanic and Atmospheric Administration & National Aeronautics and Space Administration, 2007; after Stöckli, Vermote, Saleous, Simmon & Herring, 2005)

The projected image of the map can then be sliced into tiles with a standard size depending on the required zoom level of the map. This is referred to as a tile pyramid because, as the zoom level increases, the number of tiles required to represent the same physical area at the previous zoom level increases exponentially, as does the level of detail for each tile. Figure 6a shows how the number of tiles for a particular area increases at each zoom level; note that each tile would represent a constant number of pixels. Figure 6b shows another visualization of the tile pyramid as explained by García, de Castro, Verdú, Verdú, and Regueras (2012), where the entire earth can be represented as one 256 x 256 pixel tile at Level 0, and representing that same area requires 16 256 x 256 pixel tiles at Level 1, with a corresponding increase in feature resolution.

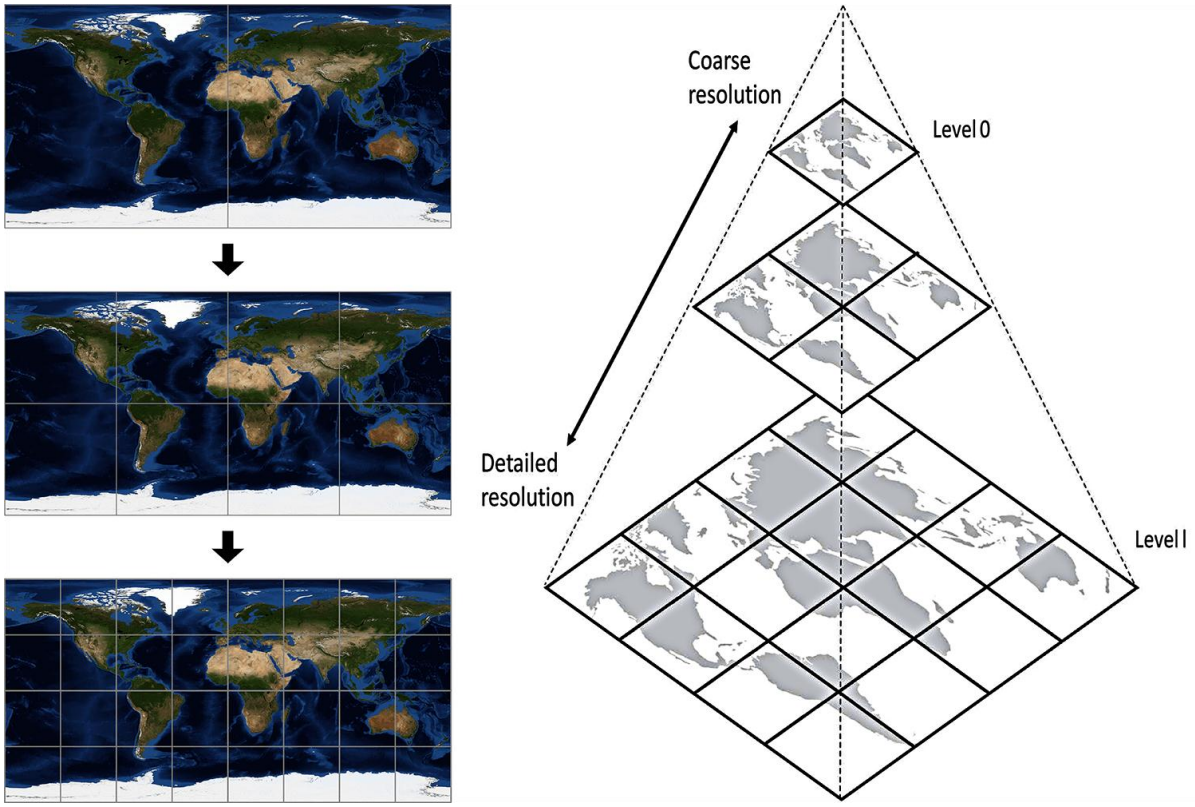

Figure 6. a (Left): Tiling of image at increasing resolution (after Stöckli et al., 2005); b (Right): Tile pyramid (from García et al., 2012)

e. Server Generated Tiles

Rendering a map tile from the raw geospatial map data requires a number of steps. The compressed geospatial map data for the entire globe is approximately 27 gigabytes. Rendering the entire globe and storing all of the resulting map tiles would require approximately 52 terabytes of storage space, most of which would be wasted, as two thirds of that space would be tiles at zoom level 18, the vast majority of which show nothing of interest (for example, open ocean where no geographically-significant features exist) ("Tile disk usage," 2011). The optimal method of serving map tiles is to render a tile on the server the first time it is requested and then to cache it for some period of time in the event that it is needed again. This prevents rendering and storing tiles that would never be requested.

In order to enable the server to request the geospatial map data for only the area to be rendered, the first step is to process the XML-based geospatial data into a spatially aware database like PostGIS (Dees & Weait, 2013). This step is processing intensive and can take hours or days but is critical to enabling the rendering engine to render only those areas of the map that are actually needed. The PostGIS database enables the rendering engine to request all of the map features that fall within a particular bounding box.

The second step is for the rendering engine to produce vector-based graphic layers for each feature using a style sheet that determines how the individual features like streets, highways, points of interest, labels, buildings, etc. should be drawn. The rendering engine then uses the painting algorithm to combine the layers into a single vector based image (Dees & Weait, 2013).

While it is theoretically possible at this point to chop the vector image into tiles and serve it to the client, this has not been widely adopted to date because browser support for SVG images has not been consistent and clipping the SVG geometry to make the tiles is challenging. There are a number of ongoing projects that are pursuing vector-based tile servers.

The next step for most rendering engines is to convert the vector image into a raster image covering the requested area plus a gutter, and then chop the raster image into standard size tiles, usually 128 x 128, 256 x 256, or 512 x 512 pixels in size, and forward them to the web server to fulfil the request.

f. Locally Generated Tiles

With the exception of proprietary formats, such as the Garmin GPS, there are very few clients that generate the map tiles on the device, for the reasons previously discussed. Most mobile devices cache the necessary tiles to show the low zoom levels, at the continent level and above, and then download the higher zoom tiles from a server as necessary. In order to locally generate the tiles, the raw geospatial data would have to be stored on the device in a spatial database and rendered into a vector image on the fly. There are a few open source

projects attempting to support this model, including Kothik JS and TileStache, but Kothik JS is still in development and TileStache is not currently designed to run on a mobile device (“Rendering,” 2014).

2. Mapping Providers

There are numerous commercial providers who serve map tiles over the Internet to support map applications, including ESRI, Google, Bing, MapQuest, Thunderforest, Stamen, CloudMade, and OpenStreetMap. These providers host map tile servers that include various styles, including aerial and satellite imagery, shaded terrain, street maps, and artistic renditions. Examples of the various styles of tiles available are shown in Figure 7.

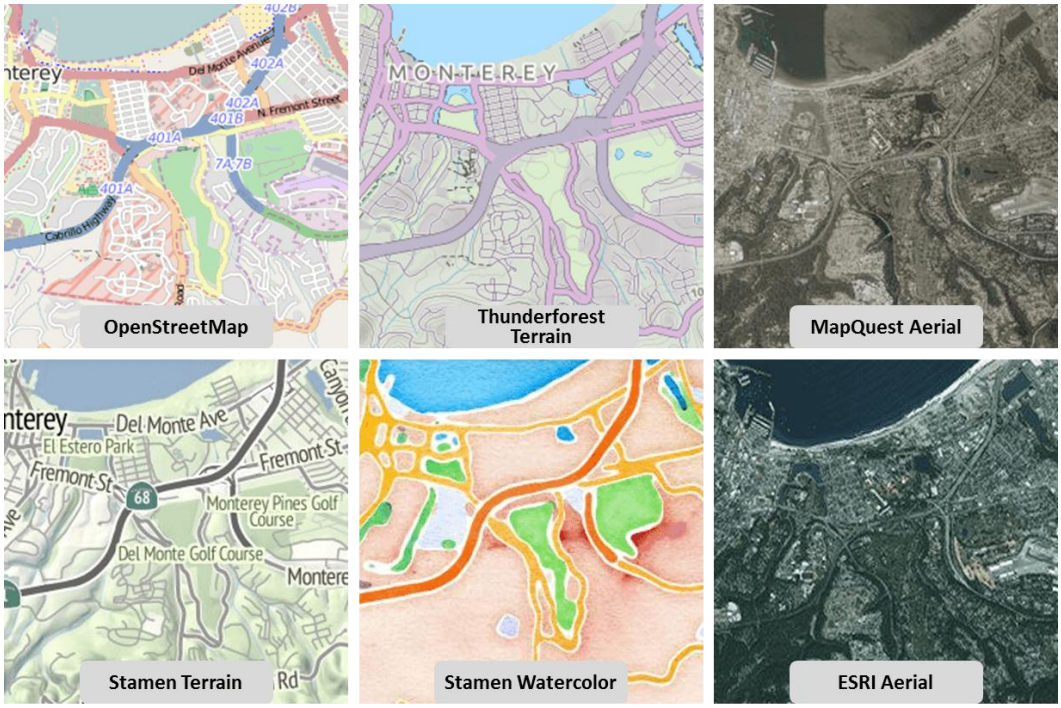

Figure 7. Tile type examples

Each of the various tile providers have different terms of service that are required to be met in order to use their tile server. All of the tile providers require applications to provide attribution of the tile source. Google and ESRI require developers to apply for an API key for usage tracking before being allowed to use

the service. Google requires developers to use the Google Maps API to access its tiles. MapQuest requires notification if anticipated usage is greater than 4,000 tiles per second. Google, ESRI, and Thunderforest track and limit the number of tiles that may be downloaded by an application before requiring a paid support plan. CloudMade is strictly a commercial provider and requires a paid support plan.

OpenStreetMap provides the base data used by several of the mapping providers. While OpenStreetMap does host a tile server for testing purposes, it prefers that application providers download the raw geospatial data and host their own tile server. OpenStreetMap provides links to software and tutorials for setting up and hosting a tile server. Hosting a dedicated tile server is an ideal solution for a military application because the tile server can be deployed close to the battlefield to alleviate the requirement for reach-back connectivity to a U.S. based datacenter.

3. Map Display Application Programming Interface

To display a map, the application must determine its location, determine which map tiles are needed, download and insert the tile images into the DOM, and handle user events such as zooming and panning. There are four popular JavaScript APIs that automate this process: the Google Maps API, Leaflet, MapBox, and OpenLayers.

a. Google Maps API

The Google Maps JavaScript API is a complete API for displaying and interacting with maps provided by Google. It includes the capability to display additional data visualization layers over the Google provided base maps. The API includes functions for adding overlays, including markers, rectangles, circles, polylines, and polygons. The Google API provides many additional services, including routing, geocoding of addresses, direction and distance calculations, weather data, information popups, and Street View pictures.

The Google API's major advantages include that it is heavily tested and is unlikely to contain many errors that would impact the use of an application. Google provides one of the best routing engines available; it is both accurate and quick to return results. As a major mapping provider, Google hosts multiple tile servers and a large user volume, so popular tiles are likely to be cached and served quickly.

There are three major limitations imposed by use of the Google API (2014). The first limitation is that it requires the developer to register an API key with Google for usage tracking. The usage tracking provides a means for Google to enforce payment for applications that exceed 25,000 map loads per day. The second limitation is that the Google API's terms of service prohibit storing the script or map tiles for use offline. This limitation means that an application using the Google API must always be used online. Such a requirement could severely impact its usage for tactical environments where continuous Internet connectivity may be lacking. The third limitation is that Google's terms of service prohibit the use of Google's API in an application that is not publicly available unless a business license has been purchased.

b. Leaflet

Leaflet is a lightweight JavaScript library, originally developed by CloudMade, Inc., that provides the capability to display and interact with maps. It is provider agnostic and is capable of interacting with many different map tile providers. It supports both vector and raster layers and includes the capability for creating overlays, including markers, circles, polygons, polylines. Leaflet is open source and extensible. There are currently over 100 plugins for Leaflet that provide additional functionality such as routing, popups, labels, heatmaps, GeoJSON layers, local file layers (KML, GPX, etc.), 3d building visualization, and various user controls.

c. MapBox

MapBox is a JavaScript library developed by MapBox, Inc., on top of Leaflet. MapBox is focused on the development of mapping technologies and software for the creation of maps. In particular they are known for developing TileMill, a map design studio, and MBTiles, an efficient database for storing map tiles locally. The MapBox Javascript library adds geocoding, interactive UTF grids, data visualization, and user interface controls to the basic functionality provided by Leaflet.

d. OpenLayers

OpenLayers is a mature, heavy JavaScript API that provides similar functionality to the other libraries discussed but is much more configurable. It supports overlays, including polylines, circles, curves, points, vector layers, and custom markers. The advantage to OpenLayers is that it is extremely configurable and supports customizing almost any part of the interface. The disadvantage is that its code size is over 700 kilobytes, which means it is not as well suited for a light-weight browser-based mobile application.

H. ROUTING

A route on a map consists of a start point, an end point, and all of the points in between that describe the path taken from the start-to-end. A route can be created in two ways: manually, with the user defining all of the intermediate points on the path; or automated, by the user defining a series of waypoints and then software determining the most efficient route that includes those waypoints.

1. Routing Algorithm

Automated routing is a shortest path problem implemented using the waypoints provided by the user along with the spatial geometry data provided as part of the map and a set of conditions. OpenStreetMap stores the geospatial data as sets of nodes, ways, and relations. The nodes are geolocated points, ways are collections of nodes that define a path or shape, and relations describe

the roles associated with their nodes or ways. Nodes, ways, and relations can all have tags that describe conditions or constraints related to them, such as speeds, turn restrictions, building type, land type, etc. Figure 8 provides an example of the way a node, a way, and a relation are described using XML. All have a unique identification number, a version, and information about the user who created the entry. A node has a latitude and longitude associated, along with tags that describe what the node is. In Figure 8, the node is the location of an exit from a highway. The way in Figure 8 is a collection of nodes, listed using the "<nd" tags, that describes a fitness center building. The relation in Figure 8 describes a right-turn-only restriction that applies to the node it references.

```
<node id="10565353" lat="33.9347502" lon="-118.1767504" version="11"
   timestamp="2011-06-11T13:57:10Z" changeset="8406172" uid="207745"
   user="NE2">
  <tag k="exit_to" v="Imperial Hwy West"/>
  <tag k="highway" v="motorway_junction"/>
  <tag k="is_in:state_code" v="CA"/>
  <tag k="ref" v="12B"/>
  <tag k="source" v="survey;image;usgs_imagery;CDOT"/>
  <tag k="source_ref" v="AM909_DSCS8452"/>
</node>

<way id="117425695" version="2" timestamp="2013-08-22T07:35:47Z"
   changeset="17450998" uid="416346" user="Brian@Brea">
  <nd ref="1322972985"/>
  <nd ref="1322972891"/>
  <nd ref="1322972954"/>
  <nd ref="1322972855"/>
  <nd ref="1322972985"/>
  <tag k="building" v="yes"/>
  <tag k="name" v="Fitness Center"/>
</way>

<relation id="1861654" version="1" timestamp="2011-11-24T19:11:01Z"
     changeset="9936279" uid="229805" user="Jim3535">
  <member type="node" ref="1515779993" role="via"/>
  <tag k="restriction" v="only_right_turn"/>
  <tag k="type" v="restriction"/>
</relation>
```

Figure 8. OpenStreetMap node, way, and relation example

Most automated routing services use an implementation of Dijkstra's algorithm, with some including performance enhancements such as bi-directional search or Contraction Hierarchies as seen in Vetter's experiments (Vetter, 2010).

The routing algorithm finds the closest point on a navigable road to each of the waypoints and then applies a shortest path algorithm to find a route between each of the waypoints subject to the restrictions described in the map geometry. Routing services that are configurable for different types of traffic (i.e., pedestrian, bicycle, vehicle) will adjust their algorithm to account for or ignore certain restrictions applicable to that mode of travel. For example, a pedestrian can travel either direction on a one-way street, while a vehicle cannot. A vehicle can travel on an interstate highway, while bicycles and pedestrians cannot.

2. Routing Service Providers

There are several commercial vendors that provide online routing services, including Google, ESRI, and HERE. All of these APIs are similar in that they take starting and destination latitude and longitude, an array of waypoints, a travel mode, and some options and return one or more routes along with turn-by-turn directions.

The Open Source Routing Machine (OSRM) is a software application that provides routing services using OpenStreetMap data. It requires pre-processing the geospatial data using a mobility profile to produce an optimized node graph (Luxen, 2014). When the server application is queried it returns the shortest path between the start and end coordinates along the OpenStreetMap road network that includes all of the waypoints by performing a bi-directional search using Dijkstra's algorithm (Luxen, 2014). It returns one or more encoded route geometries and a set of turn-by-turn directions for each route. The client must decode the route geometry into a set of coordinates that define the route. One disadvantage to OSRM is that the mobility profile is set during the pre-processing step. This means that in order to support both pedestrian and vehicle routing, there must be two servers; one for each mobility profile.

I. CONNECTIVITY CONSIDERATIONS

A mobile application can experience differing levels of connectivity depending on the infrastructure available and a given infrastructure's connectivity

to the wider Internet. The application may have high connectivity and high available bandwidth if Wi-Fi on a broadband backbone network is available. If the mobile device has been provisioned and there is cellular infrastructure, the device may have high connectivity, but bandwidth may be limited depending on the type of cellular connection (e.g., 3G vice 4G). There may also be times when there is no connectivity available at all, either due to lack of infrastructure, interference, or security restrictions. In order to be useful, a mobile application should adapt to the changing connectivity environment and continue to provide as much functionality as possible to the user.

1. Map Cache

Changing levels of connectivity present a challenge for mapping applications. The slippy map standard technique for displaying tiles uses a just-in-time methodology: each tile is downloaded only when it is visible on the user interface.

In a connected, high-bandwidth environment this works well. The download of the initial batch of tiles may take a second or two, but after the initial download delays are minimized. The number of tiles required to support a pan operation is small and zooming uses image manipulation to stretch the existing tiles to the new zoom level and then replaces them with the new tiles as they become available. In addition, the web browser will typically cache the tile images, so panning or zooming back to an area with has already been displayed is extremely responsive because there is no delay waiting on the tiles to download. These techniques make the map feel very responsive to the user.

In an environment with limited or no connectivity, the application has to maintain the map images in local (device-resident) offline storage. The advantage to storing the map data offline is that no connectivity is required and the user does not have to wait while tiles are downloaded. The disadvantage is that limitations in device storage limit the amount of map data that can be stored offline. Offline tile storage requires some method of optimizing the usage of

space and an organized method for retrieving tiles as they are needed. There are three basic methods: storing the tiles as files, using a database, or using a custom file format.

As discussed by Sample (Sample & Ioup, 2010), storing individual tiles as files is the easiest method for offline storage. Tiles are referenced by their X and Y coordinates and their zoom level. This data can be used to form a directory structure or a file name. For example, storing a tile with coordinates (X,Y) at zoom level Z using PNG format using a directory-based structure would look like *%tile_cache%/X/Y/* with the different zoom-leveled files for that X and Y coordinate residing in the directory. Using a file name format would name the files based on their X, Y, and Z coordinates and store them in the cache directory using a format like *%tile_cache%/X-Y-Z.png*. The advantage to this scheme is that it is extremely simple to find a particular tile given its X, Y, and Z coordinates, and displaying a cached tile is as simple as substituting the tile server's URL with a local file URL. There are two disadvantages to this method of local storage. Some operating systems have a limit on the number of objects that can be stored in a directory, for example, FAT32 disks popular on android only support 65,534 files in a directory and 4 million files on the device (Microsoft Corporation, 1999). Since the number of tiles increases exponentially at each zoom level, attempting to store more than a few zoom levels for any given area will easily exceed this limit; for each tile at zoom level 1, level 8 will require 65,536 tiles, given that each level doubles the tiles in each of the X and Y axes.

Custom file formats are specific to the developer or project. They may include features such as compression, storage by zoom level, clustering by location, etc. These custom formats can be optimized to the anticipated usage of the application. Clustering by zoom level or location can enable the application to uncompress an area or zoom level and pre-load it into memory in anticipation of it being used (Sample & Ioup, 2010).

Sample and Ioup (2010) also discuss the use of databases for storing map tiles because of the ease of lookup and because specially crafted databases can

use views to map multiple coordinates to the same image to improve storage. Maps that are not satellite or aerial imagery based will have large areas where tiles at multiple zoom levels are the same color (e.g., oceans, deserts, lakes, forests). In a file-based storage system, each tile would be stored, resulting in identical tiles being stored multiple times and wasting significant space. An intelligent database storage system will not store additional copies of identical tiles, but will map multiple views to a single copy of the tile. MapBox, for example, has implemented a database specification called MBTiles that stores map tiles in an SQLite database using this method (Mapbox Inc., 2014).

2. Routing

Determining a route using road networks is a shortest path problem that requires a tradeoff of either significant processing power or pre-computed graphs that require a substantial amount of space (Vetter, 2010). Performing road-network rendering on the mobile device without connectivity requires the routing graphs to be pre-computed and stored on the device. This may be a reasonable time-space tradeoff for smaller areas; Vetter noted they were able to store routing graphs for Germany, approximately 357,000 km^2 in 6.8 gigabytes (2010). A more reasonable approach may be to conduct road-network routing only when online, using a server to provide the routing such as the Open Source Routing Machine, and then to store the generated routes locally on the device. Offline routing can be performed manually by selecting coordinates for each point along the route. Manual routing, combined with a reasonably large tolerance for accuracy should produce similar results while tracking the route, with the advantage that manual routes will be smaller and more efficient to store due to fewer nodes.

3. Database Replication

Intermittent or no connectivity negatively impacts database replication, particularly in a NoSQL database that uses optimistic replication and only guarantees eventual consistency, in several ways. The longer a client is offline,

the more record differences will accumulate between that client and the other clients. Once the client does go online, the amount of data that needs to be replicated in both directions may exceed the capability of the system given the bandwidth available, in which case the client's database may never become up to date. Additionally, intermittent connectivity increases the potential for conflicting updates to the database. Resolving data conflicts quickly ensures that all copies of the database will converge to a consistent view of the data in a reasonable amount of time (Anderson, Lehnardt, & Slater, 2010). If conflicts are allowed to persist, convergence may never occur and conflicts may build to the point that they require manual reconciliation. These problems can be minimized by designing the database tables and the interaction of the clients to minimize opportunities for clients to make conflicting updates to the same record.

J. SUMMARY

This chapter provided an explanation of the problem we are attempting to solve and some of the considerations that led us to a cross-device development solution that allows our solution to target any mobile device. It explained some of the considerations for developing a mobile application that depends on mapping and discussed the advantages and disadvantages of various approaches. This discussion is intended to provide an understanding of the basic concepts used in the design and prototype implementation of our solution described in the following chapters.

III. ARCHITECTURE

A. INTRODUCTION

This chapter explains the system architecture that was developed to implement the prototype handheld assistant. It provides an overview of the entire system, a detailed explanation of the COTS components that support the mobile application, and a general breakdown of the major components of the mobile application.

B. SYSTEM ARCHITECTURE OVERVIEW

The system can be broken into three major components, as described by Figure 9: the mobile application, the supporting components, and external databases. Network and connectivity support as indicated by the lightning bolts in the diagram is handled by the mobile device operating system.

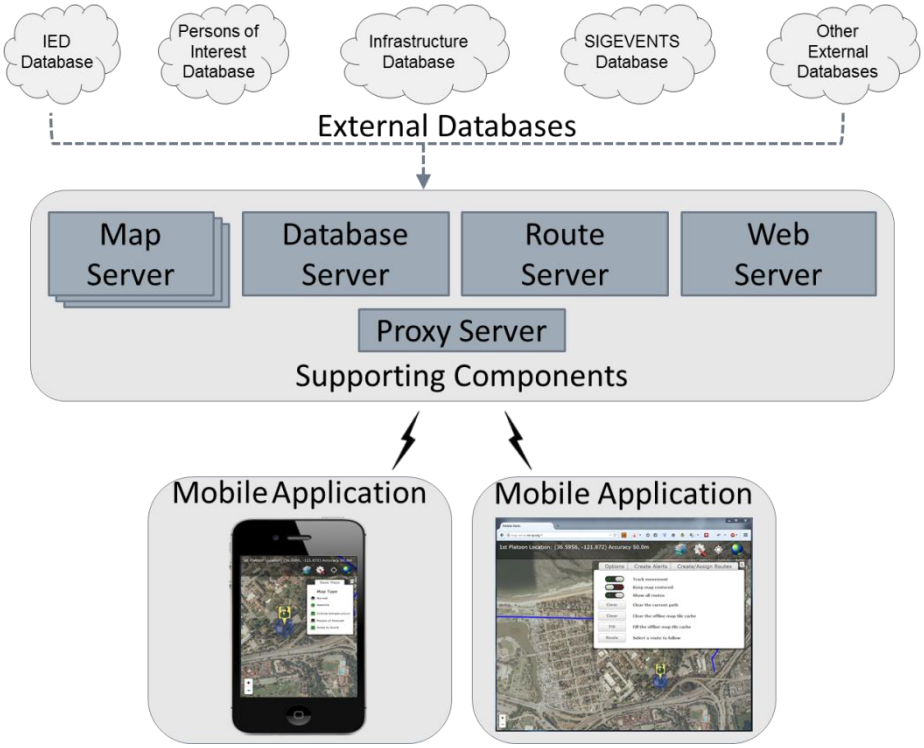

Figure 9. Overall system architecture

The mobile application is a hardware independent, single-codebase application that includes all of the functionality to support providing alerts to the user based on the supported mission. The supporting components include a map server, a database server, a web server, a route server, and a proxy server. The external databases are any existing database that contains data relevant to the users' mission that can be geospatially located and imported into the system to provide alerts.

C. EXTERNAL DATABASES

The external databases that are used are dependent on the mission to be performed and the activity using the application. A police organization might have a database that tracks gang information, such as gang territory, key personnel and their addresses, and gang-related incidents. It might also be expected to have databases that track recent criminal activity, traffic incidents, domestic incidents, and repeated calls for service. A military organization in a combat zone might have databases that track improvised explosive device (IED) events, persons of interest, key civilian contacts, and recent enemy activity. A unit conducting humanitarian relief in a disaster zone would likely have databases that have information about key infrastructure, medical support, logistics depots, and locations that have or have not been searched.

In order to use these databases with the prototype handheld assistant, queries must be created that take the relevant information from each database and translate it into a form that can be used by the mobile application. The complexity of the required queries will depend on the type of database and schema, and will be different for each external database. While we did determine the types of data that should be available in the prototype application, we did not perform a comprehensive review of all of the different possibilities for external databases; such would be necessary to move toward a production system.

D. SUPPORTING COMPONENTS

The supporting components form the glue that ties different instances of the mobile application together. When the mobile application is online, it uses the servers that are part of the supporting components to exchange data and acquire map tiles. Separate boxes in the architecture diagram represent the individual servers in the supporting components because they are modular. For the prototype, each of the server components is implemented with COTS software that provides a basic set of defined functions. This provides the ability to swap any of these components with one from a different vendor with few, if any, changes to the mobile application.

1. Map Server

The map server provides the map tiles that the mobile application displays to inform the user about relevant locations. In the architecture diagram, Figure 1, the map server has multiple boxes because the mobile application is designed to support a multitude of different map providers. The map server component must only fulfil two requirements to work with no changes to the mobile application: it must serve tiles using the EPSG3857 coordinate reference system and a spherical Mercator projection, and it must properly set the HTML cross-origin request headers when it responds to tile requests.

We tested the prototype application with several commercial map servers, as well as a locally built OpenStreetMap tile server. The application, as built, supports three different base maps sources: MapQuest aerial view, ESRI world imagery, and our OpenStreetMap server. As discussed in Chapter V, the mobile application could be extended to support any number of servers as part of future work. To address a map server, the application must be provided with a URL, as in Figure 10, that contains the logical addressing scheme for the tiles, where {s} is a server number, and {x}, {y}, and {z} are the x, and y coordinates and zoom-level of the requested tile.

MapQuest Open Aerial Imagery
http://oatile{s}.mqcdn.com/tiles/1.0.0/sat/{z}/{x}/{y}.jpg

Esri
http://server.arcgisonline.com/ArcGIS/rest/services/World_Imagery/MapServer/tile/{z}/{y}/{x}

OpenStreetMap Server
http://*%server%*/osm_tiles/{z}/{x}/{y}.png

Figure 10. URL addressing scheme

Most of the commercial map tile providers that serve tiles over the internet like ESRI, OpenStreetMap, CloudMade, Google, etc. limit the number of free tiles that may be accessed in a given time frame, either by IP address, or by requiring the use of an API key that must be sent to the server upon initiating a connection. Applications that access tiles in excess of the provider's limit for free tiles are either blocked or assessed a usage fee. For the prototype, we setup a local OpenStreetMap server to support testing to prevent having to worry about commercial provider limits. However, consideration must be given to usage-based service as part of a support plan cost for a production system.

Our OpenStreetMap server has five main components as described on the Switch2OSM webpage:

> Mod_tile, renderd, mapnik, osm2pgsql and a postgresql/postgis database. Mod_tile is an [A]pache module, that serves cached tiles and decides which tiles need re-rendering—either because they are not yet cached or because they are outdated. Renderd provides a priority queueing system for rendering requests to manage and smooth out the load from rendering requests. Mapnik is the software library that does the actual rendering and is used by renderd. (Dees & Weait, 2013)

The software used by the OpenStreetMap server is designed to run on Linux, so to enable all of the supporting component servers to run on one Windows computer, the server was setup in a virtual machine loaded with Ubuntu 14.04. The server was configured as described on the Switch2OSM webpage and loaded with the current OpenStreetMap data for the northern California area and serves tiles to the mobile application upon request.

2. Database Server

The database server maintains the databases that are used to hold alert data, manage routes, communicate between units, and track the location of the different units using the mobile application. We chose a NoSQL, document-based database because of the unique challenge of integrating date from multiple external databases where data for different types of alerts may not be relational. A document-based database treats each individual 'record' as a stand-alone document. Unlike in a relational database, while multiple documents may share some of the same fields, there is no requirement that they do so. For example, Figure 11 shows the records for two different alerts in the database. Some fields are required for the document to be useful: the title, location, type of alert, etc., but other fields may vary depending on the subject of that document. Document 2 has a field called "warn_radius" that describes the size of the circular alert area in meters. Document 1 has no need for a "warn_radius" because it describes an irregularly shaped area defined by the points in its location field. Document 2 also has multiple attachments, while document 1 only has one.

Document 1


```
{
        "_id" : "d155b07ef4cc9ba33f3158f29d001e47,"
       "_rev" : "3-2da8c9a02ca573d9d24f526a3a1cd94e,"
  "characteristics" : [{"Start date": "1 May 2014"},
            {"End date": "5 August 2015"}],
    "description" : "This area is under construction and
            should be avoided during this
            period.,"
     "location" : [ [ 38.871839, -77.055411 ],
              [ 38.872298, -77.054225 ],
              [ 38.872946, -77.054655 ],
              [ 38.872829, -77.055588 ] ],
   "location_type" : "irregular,"
       "title" : "Area Construction,"
   "_attachments" : {
 "under-construction.png": {
     "content_type" : "image/png,"
        "digest" : "md5-hhyKSWUP4Q88+iCzq04QdQ==,"
        "length" : 50440
        }
    }
}
```

Document 2


```
{
       "_id": "d155b07ef4cc9ba33f3158f29d0009b3,"
      "_rev": "8-ae24034fcf58a096fa8917937b676069,"
 "characteristics": [ { "Length": "0.25 miles" },
           { "Load Capacity": "25 tons" },
           { "Road Condition": "Moderate" } ],
   "description": "The Cambridge Street Bridge is critical
           to maintaining logistics support for
           operations on the East side of the
           river.,"
    "location": [ 38.869614, -77.061058 ],
  "location_type": "circle,"
      "title": "Cambridge Street Bridge,"
   "warn_radius": 80,
  "_attachments": [{
       "Photo1.jpg": {
      "content_type": "image/jpeg,"
         "digest": "md5-jV2roqr/pD57GCA+JrhxmQ==,"
         "length": 8249
         },
       "Photo2.jpg": {
      "content_type": "image/jpeg,"
         "digest": "md5-jV2roqr/pD57GCA+JrhxmQ==,"
         "length": 4685
  }]
  }
}
```


Figure 11. Example alert documents

This format is useful because unlike a relational database, additional data fields can be added to a document without having to adjust the schema for every other document in the database. Each field is setup as a key – value pair, where the key is an alphanumeric string, and the value can have any type representable in JavaScript Object Notation (JSON), which makes it easy to parse directly into a JavaScript data structure in the mobile application.

Apache's CouchDB was chosen as the database system for the prototype application. CouchDB is an open source, document-oriented, NoSQL database with an HTTP REST interface. It stores documents using a JSON compatible format and supports bi-directional replication via HTTP. As a NoSQL database, there are three ways to query documents: by their unique document identifier, by retrieving all documents, or by using a map/reduce function. A map/reduce function is a NoSQL database programming model that uses a map function to filter and sort documents, and a reduce function to perform calculations such as counting the number of records with a particular value.

A unique advantage of CouchDB is that there is a JavaScript library, PouchDB, that provides a local database on any web-enabled client that can be synchronized with an online CouchDB instance. This enables an application to work with a local copy of the database when offline, and then synchronize everything when connectivity becomes available. PouchDB is discussed in more detail in Chapter IV.

3. Route Server

The route server provides an automated method for creating a route. The application queries the route server, providing start and end coordinates, along with any intermediate waypoints, and the route server calculates the shortest route or routes from the start point to the destination, including the waypoints, using the road network. The route server responds to the query with a JSON object that contains a route (if one was found), and depending on the provider, may include route geometry, road or traffic information, turn-by-turn directions, or alternate routes. There are several commercial providers of routing services that follow this model including Google, MapQuest, Microsoft, ESRI, and HERE.

Our prototype system uses the Open Source Routing Machine (OSRM) software module, which is:

> a C++ implementation of a high-performance routing engine for shortest paths in road networks. It combines sophisticated routing algorithms with the open and free road network data of the OpenStreetMap (OSM) project. OSRM is able to compute and output a shortest path between any origin and destination within a few milliseconds. (Luxen, 2014)

We installed OSRM on the same virtual server as the map server and configured as described on the OSRM Wiki. The routing preferences were configured for automobile traffic; in the event pedestrian traffic is anticipated, the routing preferences would need to be updated and the OSRM node graph would need to be recomputed. When the mobile application is online, it can query the route server as described above and receive a response that includes encoded

route geometry, turn-by-turn directions, and alternate routes. If the application is not online, the automated routing functions are not available.

4. Web Server

Our mobile application is written in HTML5, CSS3, and JavaScript. It can be compiled and run as a Cordova application on a phone, tablet, or other mobile device, or it can be run as a web application in a browser on a laptop or desktop computer. While it is possible to run the application by loading the index.html file directly in a browser, some browsers impose additional cross-origin security restrictions on pages loaded via “file:” URIs which can break the application. Our prototype system uses a simple Python web server to enable access from laptop or desktop computers. The application requires that these computers have network access to the web server.

5. Proxy Server

The proxy server aids with testing and demonstrating the prototype system. It is a simple nodeJS script that enables the database server, map server, route server, and web server to be located at the same host address. The proxy server listens on the standard HTTP port 80, determines for which server process the request is destined based on the format of the URL, and forwards it to the correct process. In addition, the proxy server sets the cross-origin access headers on all responses, if the server process had not already set them correctly. The proxy server could be eliminated in a production system, as each of the server hostnames/IP addresses are independently specified in the application’s code.

E. MOBILE APPLICATION

The application is a web-based application that runs on top of Cordova-compiled native code that interfaces with the mobile operating system, as depicted in Figure 12. Cordova also provides plugins that are compiled as part of the application to provide JavaScript access to the device’s hardware. The focus

of our effort is on developing the web-based portion of the application that is designated as “Mobile Application” in Figure 12.

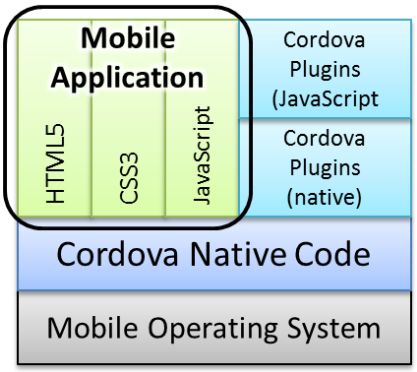

Figure 12. Mobile application architecture

The code for the web application can be broken down into five major sub-systems, each of which supports specific functionality within the application. The sub-systems are user interface, mapping, database, routing, and cache. A block diagram of the sub-systems and their interaction is shown in Figure 13. An overview of the sub-systems is provided here and a detailed explanation of their implementation is included in Chapter IV.

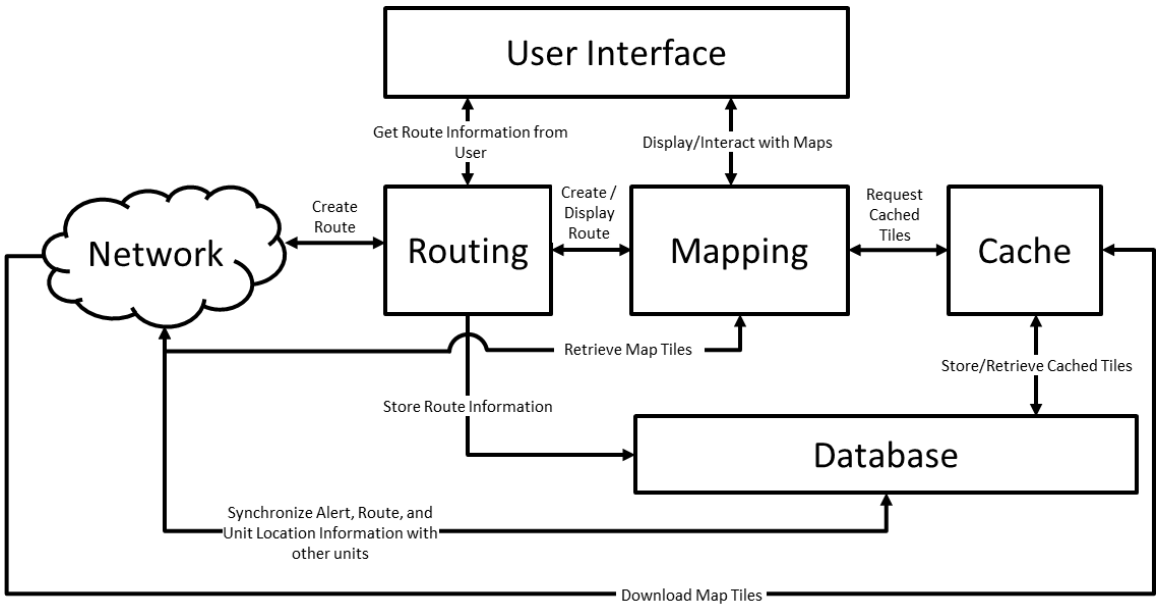

Figure 13. Subsystems block diagram

1. User Interface

The user interface sub-system manages all of the application's interaction with the user. It handles displaying and hiding various dialog boxes, re-sizing of the display in the event of device rotation or window resizing, and managing user events like touch and mouse interaction.

2. Mapping

The mapping sub-system manages displaying, zooming, and panning of the map, geolocation, and displaying other unit locations. It manages the display of alert areas, tracks the user's location, and notifies the user in the event he enters an area that requires an alert be generated.

3. Database

The database sub-system creates the local database if it does not exist or opens it if it exists when the application is started. It connects to and synchronizes with the master database if the application is online and then establishes a replication schedule for each of the pertinent databases. It

interfaces with each of the other systems, providing other unit locations to the mapping sub-system and route assignment information to the routing sub-system.

4. Routing

The routing sub-system provides the application the capability to create both automated and manual routes. It also provides the ability for a user to choose to follow a route and be tracked along that route. It enables a unit to assign a route to another unit, uses the database subsystem to coordinate the assignment, and then notifies both the assigned and assigning unit if the assigned unit deviates from the route.

5. Cache

The cache sub-system handles the caching of map tiles for use when the application is offline. When directed by the user, the cache system downloads and stores all of the map tiles for a given area around the user's current location, within the limits of the mobile device's storage. The cache sub-system then monitors the application's network connectivity; in the event connectivity drops, the cache sub-system seamlessly switches the mapping sub-system onto the local cache of map tiles.

F. SUMMARY

This chapter has explained the system architecture, the purpose of each of the COTS supporting components and how they were configured, and the different sub-systems of the web application's code and how they fit into the Cordova mobile application. The next chapter explains how the different sub-systems in the mobile application were developed, the functions that the application can perform, and the testing that was accomplished.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. IMPLEMENTATION

A. INTRODUCTION

This chapter discusses the overall design of the application, the breakdown of the different functions in the application and the reasoning behind how and why each one was implemented, and the algorithms used for tracking along a route and caching map tiles. In addition, the user interface and each of the application's user functions is described. The chapter concludes with an explanation of the testing performed.

B. APPLICATION DESIGN

Our mobile application is written in HTML5, CSS3, and JavaScript. It has approximately 4,000 lines of code, with an additional 20,000 lines of code in supporting open source libraries. The overall size of the application is eight megabytes. It consists of a main HTML file that is loaded by either the web browser or by Cordova. That HTML file contains the majority of the user interface structure and loads six CSS files that define the user interface styling, a main JavaScript file that supports the overall program flow, and 16 additional JavaScript files that support specific functions. As discussed in Chapter III, the mobile application's code can be broken down into five modular and loosely coupled sub-systems: user interface, mapping, database, routing, and cache.

1. User Interface

The user interface is created with the main HTML file, which generates all of the DOM objects for the main map window, the status bar at the top of the screen, the icons, and most of the dialog box windows. The dialog boxes are shell objects that are created, but not displayed when the program is run. When a dialog box needs to be displayed, the application fills the appropriate HTML into the dialog box shell and then changes the display style from "none" to "inline." This allows a dialog box element in the DOM to serve multiple functions just by

changing its inner HTML content. The user interface sub-system includes all of the code to set the control states in the options and layers menus, to display and hide status messages, to re-size the user interface on screen rotation, and to change the GPS and network icons depending on the reported hardware state.

The user interface is structured with CSS3, which supports all of the animations, buttons, and dialog box sizes and styles. All of the elements are styled using “em” units instead of pixels. These units, derived from typesetting traditions, are based on the horizontal size of the font assigned to the body element of the page (Lie & Bos, 2005). By basing the size of all elements on the page proportionally to the base font size, it becomes trivial to scale the dialog boxes and other parts of the user interface to fit any screen size by manipulating the base font size. This makes sure that all elements resize properly and retain their proportions on devices with varying screen resolutions. The application attaches a handler to the window resize event that gets called any time the browser window changes size or shape. The handler checks the screen width and height and then adjusts the base font size proportionally and forces a browser reflow/repaint. This ensures that all dialog boxes are redrawn at the correct size and location anytime the mobile device screen is rotated or the browser window is resized. If we did not do this, a screen rotation or window resize might cause a dialog box to extend off-screen, which would cause the browser to add scroll bars and disrupt the “application” experience.

2. Mapping

The open source Leaflet library, authored by Vladimir Agafonkin (2014), provides the map interface. The Leaflet API enables the application to display the map, switch base layers, add markers and other shapes, change the zoom level, and manipulate the map in various ways. The application makes use of several Leaflet plugins to add additional features, including displaying labels for the unit markers and routes, and for creating the adjustable polyline used when creating a manual route.

3. Database

The application uses the open source PouchDB library to support all of its database functions. PouchDB is a JavaScript library designed to “provide a unified abstraction layer over other databases” using a CouchDB compatible API and to seamlessly replicate between an online CouchDB database and a local PouchDB database (Harvey & Lawson, 2014). PouchDB is browser agnostic, supports SQLite, WebSQL, IndexedDB, LocalStorage, and LevelDB as backend databases, and will select the appropriate backend database depending on what is available on the device. Documents can be queried by their document id, by requesting all documents in the database, or by using map/reduce functions. All documents are required to have a unique document id that can either be assigned by the database upon document creation or can be included as part of the document when creating it. All database operations are asynchronous and require the use of a callback function in order for the application to be notified once the operation is complete.

4. Routing

The routing sub-system consists of three major sections: automated route generation, manual route generation, and checking whether or not a unit that has been assigned a route is on it.

a. Automated Route Generation

The OSRM server discussed in Chapter III generates automated routes. The class that provides the application’s interface to the OSRM server is the Leaflet Routing Machine developed by Per Liedman (2014). Once the user generates a set of waypoints by clicking on the map, the set is passed to the Leaflet Routing Machine class, which formats a request to the OSRM server. The class then interprets the response from the server, draws the route on the map, and provides a control that shows the turn-by-turn directions and any alternate routes. The user can manipulate the auto-generated route by adding waypoints or dragging generated waypoints to different locations, thereby causing the class

to re-query the server and adjust the displayed route. Once the user is complete, the class returns a set of points that define the route geometry that the application stores in its routes database.

b. Manual Route Generation

Manual route generation uses a Leaflet Plotter class developed by Nathan Mahdavi (2014). The class enables the user to add waypoints to the route by clicking on the map. Existing waypoints can be removed by clicking on them and the Plotter class removes that point and re-draws the route. This class also enables waypoints to be dragged around the map to adjust their position. Once the user is finished, the class returns an array of coordinates that defines the route geometry.

c. Route Checking

Once the user is assigned or elects to follow a route, the application has to determine whether the user is on the route or not. It accomplishes this by adding a callback function that is executed each time the user's geolocated position is detected as having changed. The callback function determines the closest point on the route to the geolocated position and whether that point is inside a specified accuracy distance using the following algorithm:

A route is defined by a set of coordinates that specify the endpoints of each segment of the route. Search through all of the coordinates that define the route and find the coordinate or set of coordinates with the shortest distance between that coordinate and the user's geolocated position. This calculation is accomplished using the haversine formula, Equation (1), which computes the great-circle distance between two points on the Earth's surface. Since the Earth is not spherical it uses an approximation of the Earth's radius and can be expected to provide a result with an error smaller than 0.5 percent (Chamberlain, 1996).

d: great circle distance

R: Earth's radius

ϕ_1, ϕ_2: latitude of points 1 and 2 (1)

λ_1, λ_1: longitude of points 1 and 2

$$d = 2R\arcsin\left(\sqrt{\sin^2\left(\frac{\phi_2-\phi_1}{2}\right)+\cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\lambda_2-\lambda_1}{2}\right)}\right)$$

If the shortest distance between the user's position and the coordinate or coordinates in the set is less than the specified accuracy distance, then the user is in a position similar to Figure 14, where L is the user's location, r is the specified accuracy distance, and P_x is a coordinate along the route. The callback function returns P_x and the haversine distance between L and P_x, and the user is considered to be on-route. In the event there are multiple coordinates in the set, only the first is used.

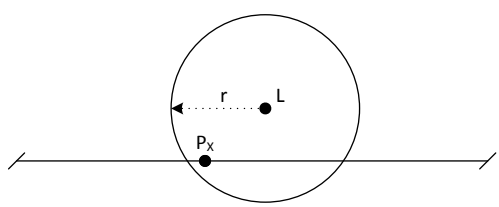

Figure 14. Position is within r distance of P_x

If the shortest distance between the user's position and the coordinates in the set is greater than the specified accuracy, the algorithm must check for an additional condition: whether the user's location is between two coordinates, but close enough to the route segment that joins them to be considered on-route as in Figure 15.

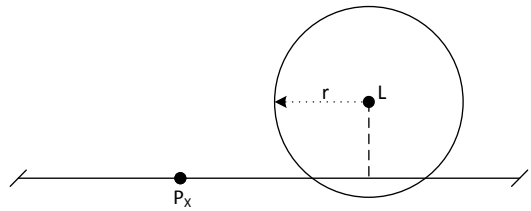

Figure 15. Position is within r distance of the route, but not P_X

To find the distance in this condition, the tangent distance t between the user's location L and the tangent point P_T on the route segment must be found. The algorithm checks the route segment on both sides of each coordinate in the set. Since the spherical geometry solution for determining tangent points is computationally intensive, for this part of the algorithm, the coordinates are transformed into Cartesian screen coordinates at the map's maximum zoom level, which allows us to use simple vector math. Consider a route segment with two end points, P1 and P2, and the user's location, L. Two cases must be examined: Figure 16, where the tangent point, P_T, is on the line segment, and Figure 17, where the closest point on the segment is one of the segment's endpoints.

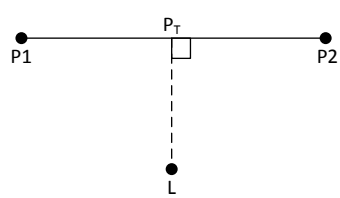

Figure 16. P_T is on the route segment

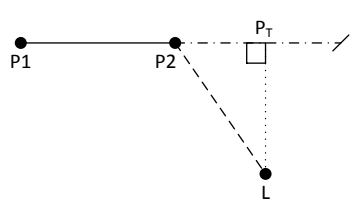

Figure 17. P_T lies outside the route segment, P2 is the closest point on the segment

Equation (2) is used to find the projection distance along the segment from P1. If k is less than zero or greater than one, Figure 17 applies and one of the endpoints is the closest to P3. If k is between zero and one, Figure 16 applies and P_T is found using Equation (3). The distance is then calculated by transforming the coordinates back into geographic coordinates and finding the haversine distance as discussed previously. If the distance is within the specified accuracy distance, the user is considered on-route.

$$k = \frac{\left[\left(P3 - P1\right) \bullet \left(P2 - P1\right)\right]}{\left|P2 - P1\right|^2} \tag{2}$$

$$P_T = P1 + k\left(P2 - P1\right) \tag{3}$$

5. Cache

The code that caches the map tiles is modified from an OfflineTileCacher class developed by Greg Allensworth as part of his Mobile Map Starter project (2014). Allensworth's class is designed specifically to interface with the Leaflet TileLayer class and uses the method discussed by Sample and Ioup (2010) of storing each map tile as an individual file using the HTML5 FileSystem API. While the HTML5 FileSystem API is well-supported by Cordova via a standard plugin, support in several desktop browsers is lacking, and Mozilla has publicly stated that they may never support the full FileSystem API (Sicking, 2012). For this reason, we modified the OfflineTileCacher class to check for Cordova upon instantiation and then use the FileSystem API if Cordova is present, otherwise store the tiles as blobs in a PouchDB database.

The class requires each map layer be registered with the cache class so that it can determine the online and offline URL that Leaflet will use to access the tiles. It develops the offline URL using the format *{Layer Name}-{z}-{x}-{y}.{extension}*, for example: *MySampleLayer-5-120-400.png*, which makes reference and storage straightforward, whether using a file or a blob. Once a tile layer has been registered with the OfflineTileCacher class, the seed method can be called, which creates a list of tiles needed to form

a tile pyramid around a given location, to download and store the tiles. The original download functionality in the class downloaded each tile sequentially, waiting until each download was complete before starting the download for the next tile. This function was modified to queue all of the required downloads immediately and allow the browser to parallelize the downloads. This maximizes the amount of bandwidth used while minimizing the time spent waiting for the cache to fill. The OfflineTileCacher class provides a function that switches Leaflet between the online and offline URLs that were determined when the layer was registered.

The cache class's interface with Leaflet occurs via Ishmael Smyrnow's FunctionalTileLayer class (2014). The FunctionalTileLayer allows the URL being passed to Leaflet to be defined and returned by a given function. The given function is executed each time Leaflet requests a tile, and since FunctionalTileLayer supports JavaScript's notion of a 'promise,' this means that the URL can be provided by a process that requires an asynchronous callback. For example, when using the offline cache in a web browser, Leaflet calls for a particular tile, the function provided to FunctionalTileLayer queries the PouchDB Tiles database, and returns to Leaflet a promise that will be fulfilled by the callback function from the PouchDB "get" method. The callback function creates a data URI from the blob returned by the database, fulfils the promise, and Leaflet is handed that data URI so it can display the map tile.

Since the map tiles are stored as individual files on the device, the number of tiles that are required to represent a given area becomes relevant given there is a limit to the amount of space available on the mobile device's file system. The formula for determining the radius of tiles at a particular zoom level that should be cached is given by Equation (4) and assumes that the number of tiles displayed at $zoom_{min}$ covers the entire area that should be cached; that is, $zoom_{min}$ is the maximum level the user can zoom "out" from the center point. This is a critical point, because if the user is mobile and $zoom_{min}$ is too large, then it will be easy for the user to travel outside of the cached area. The formula

provides an edge of two tiles around the center tile, which gives $(1 + 2 + 2)^2 = 5^2 = 25$ tiles at $zoom_{min}$. To cache a larger area, the two in the equation can be changed, having a predictable impact on the number of tiles that will be downloaded.

$$edge = 2\left[1 + \left(zoom - zoom_{min}\right)\right] \tag{4}$$

The amount of disk space required is given by Equation (5), where $\bar{t}_{size}$ is the average tile size.

$$size_{disk} = n_{tiles} \cdot \bar{t}_{size} \tag{5}$$

The estimated download time is given by Equation (6), where $size_{disk}$ and $rate_{DL}$ are in bytes.

$$t_{DL} = \frac{size_{disk}}{rate_{DL}} \tag{6}$$

Table 2 shows the number of tiles, estimated download size, and anticipated download time required to cache each zoom level starting with an arbitrary $zoom_{min}$ of level X. The average tile size used for the Table 2 was determined experimentally as 8,072 bytes. The download rates used in Table 2 assume the theoretical maximum bandwidth at the physical layer for each media type and does not consider protocol or data link layer overhead. While real-world results will necessarily be worse, this gives us a best case figure for the expected download times. This data shows that the number of layers being cached will have a significant impact on storage space and bandwidth use during the cache download process. For this reason, the number of layers to be cached is limited programmatically. When the application switches to use the offline cache, the ability to zoom the map is programmatically limited to the layers that were cached to prevent the user from zooming to a layer that is not in the cache.

Zoom Level	X	X+1	X+2	X+3	X+4	X+5	X+6	X+7	X+8	Total
Radius	2	4	8	16	32	64	128	256	512	N/A
Tiles Required	25	81	289	1,089	4,225	16,641	66,049	263,169	1,050,625	1,402,193
Download Size (including headers)	201.80 Kb	653.83 Kb	2.33 Mb	8.79 Mb	34.10 Mb	134.33 Mb	533.15 Mb	2.12 Gb	8.48 Gb	11.32 Gb
Time to Download Each Layer(Calculated) in Seconds										
802.11n (600 Mbps)	0.0026907	0.0087178	0.031104	0.117205	0.454723	1.7910154	7.1086337	28.324	113.0753	150.913
3G (384 Kbps)	4.2041667	13.6215	48.60017	183.1335	710.5042	2798.4615	11107.24	44256.25	176680.1	235802.12
Edge (1894 Kbps)	0.8523759	2.761698	9.853466	37.1295	144.0515	567.37551	2251.9431	8972.757	35821.1	47807.8
4G (326 Mbps)	0.0049521	0.016045	0.057247	0.215716	0.836913	3.2963473	13.083375	52.13007	208.114	277.75
802.3 Gigabit Ethernet (1Gbps)	0.0016144	0.0052307	0.018662	0.070323	0.272834	1.0746092	4.2651802	16.9944	67.84516	90.54

Table 2. Tile count, estimated size, and anticipated download time by zoom level

C. APPLICATION FUNCTIONALITY

The user interface and application functionality is very similar on all devices, so that a user who understands the interface on one device can move seamlessly to another without additional training. This section provides an overview of the capabilities of the application. Some of the application's capabilities are purposefully limited on mobile devices due to their smaller screen size compared with tablets or laptops; where this is the case, it is identified. In cases where the interface and functionality are the same, figures in the text are from representative devices on which the application was tested.

1. Main Interface

The main interface for the application is extremely simple, as depicted in Figure 18. Along the top of the screen is a transparent status bar that provides the user's current location in latitude and longitude, the accuracy reported by the GPS, and any status messages. On the right side of the status bar are four large icons. The first icon brings up the layers menu, the second icon brings up the options and actions menu, the third icon indicates the GPS status, and the fourth icon indicates the network status.

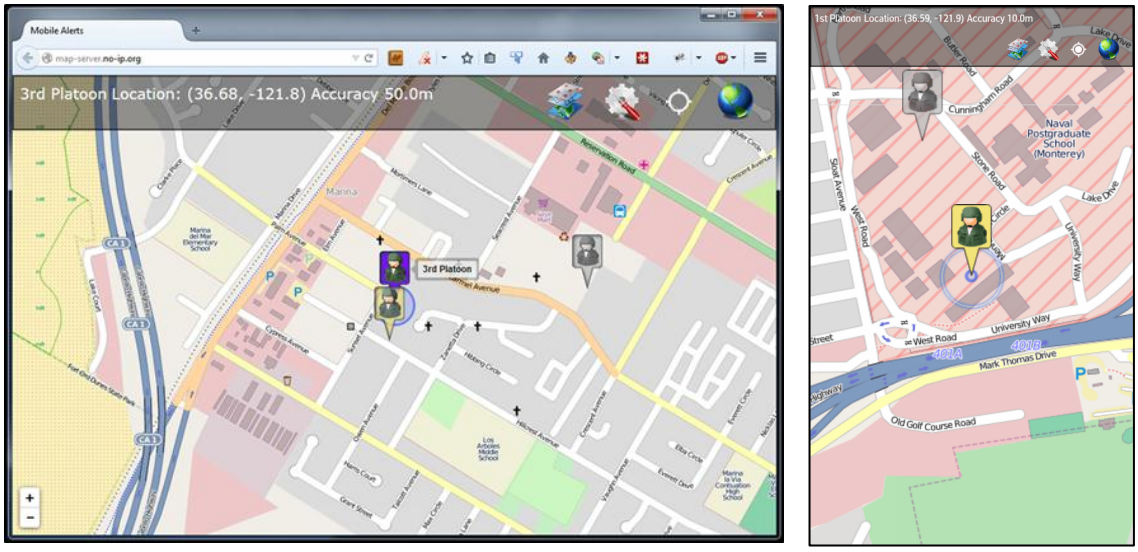

Figure 18. Main interface: browser (left), Android phone (right)

If the operating system indicates no GPS device, the user will be notified and the application will terminate. If a GPS is present but cannot get a location, such as if the user is indoors, the application will display a “Waiting for GPS” message indefinitely until a location is determined, as shown in Figure 19. The application must obtain at least one position response from the GPS or other location provider before it can display the map. The center of the GPS icon will be a filled circle if the last GPS query returned a valid result, and will be an empty circle if the last GPS query resulted in failure to get a location.

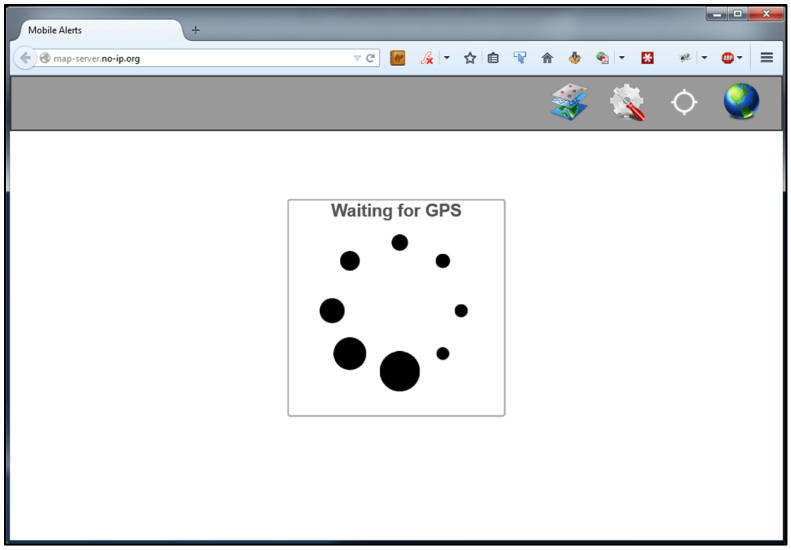

Figure 19. Waiting for the GPS to become available.

The Network icon is a globe that indicates whether the application is online, Figure 20. If the application is on a mobile device and is able to determine the type of connection (e.g., Wi-Fi, 3g, 4g) then the application displays the connection type on top of the globe. If the application does not have any network connectivity then the globe is greyed out. In most desktop browsers the icon appears in color even if the workstation has no Internet connectivity because the application relies on whether the browser reports its online status accurately, and most do not.

Figure 20. Icons for offline, online type unknown, Wi-Fi, 3g, 4g (after Bu, 2011)

Beneath the status bar is the map display. When the application runs in a browser the map includes buttons to zoom in and out, while on a mobile device zooming in and out can be accomplished by using the standard pinch gesture.

The map can be panned in any direction using the mouse or touch gestures. If the option to keep the map centered on the user has been enabled, the map re-centers each time a GPS position report is received.

The application shows the user's location in color at the center of the map screen with a blue pulsing accuracy ring around it. The accuracy ring is to remind the users that although the icon is drawn at the GPS provided latitude and longitude, their actual position could be anywhere within that ring. Other units whose location information is in the database will also be plotted on the map screen. Each unit is assigned a random unique color by the application when that unit is created and added to the database. Each time a unit changes position, its database entry is updated with its current location and a time stamp. When the application draws other units on the map, if their location is current within the past two minutes, the unit is drawn using its unique color. If the database timestamp for a unit has not been updated within the past two minutes, the unit's icon is drawn in grey to indicate that the location presented is the last known location, but that it may not be accurate any longer. This behavior is shown in Figure 21, where 1st Platoon is the unit whose display is being shown, 2nd Platoon has not had a position update in the past two minutes, and 3rd Platoon has provided an up-to-date position report.

Figure 21. Recent and non-recent position reports for other units

2. Layers Menu

The layers menu is accessed via the layers icon, as depicted in Figure 22, and has two tabs, each of which is dynamically generated at application run-time.

Figure 22. Layers Menu icon (Icons-Land, 2014)

a. Base Maps Tab

The Base Maps tab, Figure 23, is built from the different base map options that have been configured. These base maps are currently configured in the application's code, but could be moved to a configuration file in the future. Since only one base map can be selected at a time, the base maps tab uses radio buttons that allow the users to select which base map they want to use on the map display. The application is currently configured to support a local OpenStreetMap server and MapQuest Open Aerial satellite imagery for base

maps. Selecting one of the base map options sets the background base map used by the map window, accordingly.

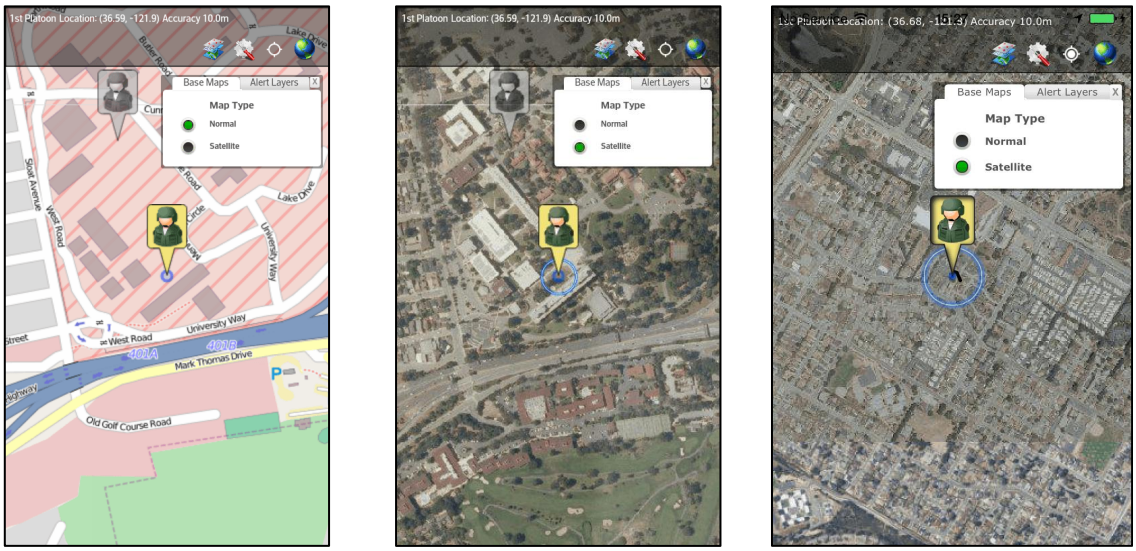

Figure 23. Base Maps tab, Android phone (left, middle) and iPhone (right)

b. Alert Layers Tab

The Alert Layers tab, Figure 24, is built based on the different categories of alerts in the database. As new categories are added to the database, they are populated to the Alert Layers tab. The database is checked each time the dialog box is shown, so if an alert type is added by another unit, it will appear on this menu once the databases replicate. The display of each layer can be turned on and off, depending on which types of alerts the user wants to see. Hiding a category of alerts has no impact on whether the user will be notified when they come in range of an alert: if the user enters the warning radius, she is warned regardless of whether that category is set to be displayed or not.

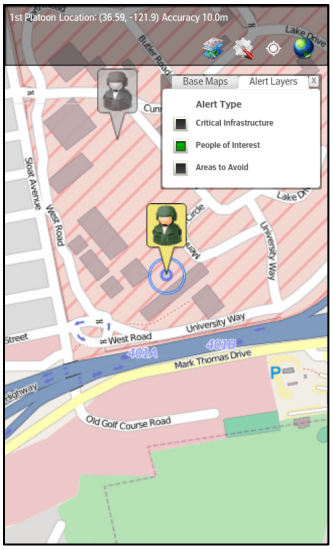

Figure 24. Alert Layers tab

3. Options Menu

The Options and Actions menu is accessed from the icon, as shown in Figure 25, and has two tabs when the application is run on a mobile device: Options, and Create Alerts. When the application is run in a browser a third tab is present, the Create / Assign Routes tab.

Figure 25. Options and Actions Menu icon (from Coelho, 2007)

a. Options Tab

The Options tab gives the user controls to change some of the options affecting how the application functions. The users can disable location tracking, for example, if they are using the application in a static location such as a command post, to monitor other unit locations. A unit that anticipates stopping in a static location for a period of time might also disable location tracking to reduce battery usage. The users can select whether or not to have the map remain

centered on their location. This is enabled by default, but can be disabled if the user wants to pan the map to view an alternate location and does not want it to re-center each time his location updates. Additional options include showing all of the routes on the map (if running in a browser), clearing the unit's current track, filling and clearing the map tile cache, and selecting a route to follow.

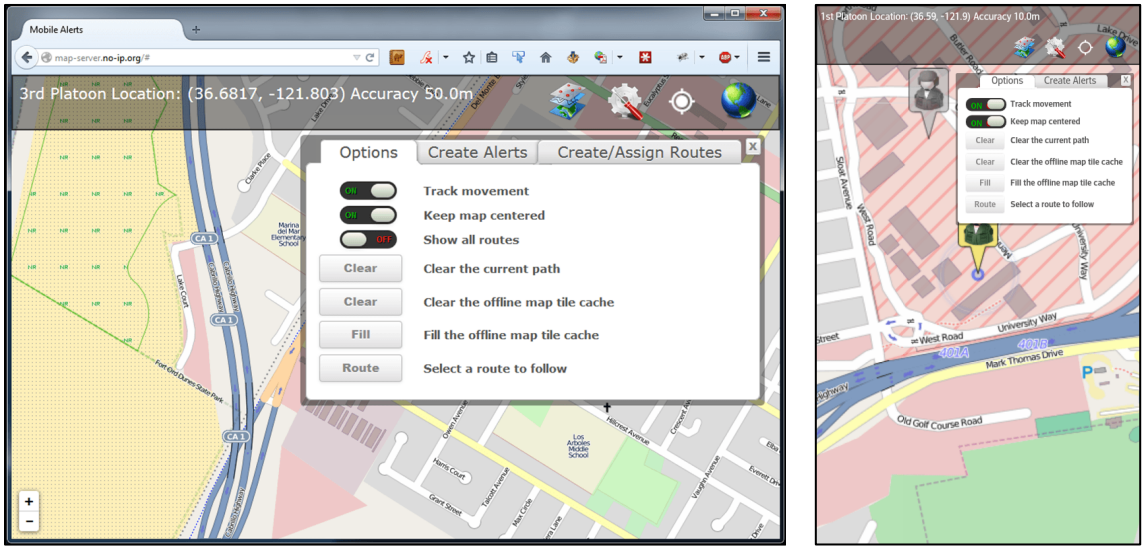

Figure 26. Options tab, browser (left) and Android phone (right)

b. Create Alerts Tab

The Create Alerts tab, Figure 27, lets a user create an alert and add it to the appropriate database. The user can choose from three different alert types: irregularly shaped, circle, and point. To create an irregularly shaped alert the user clicks on the map to define the outline of the area; for circle and point alerts the user selects the location for the center of the alert. Once the area is defined, the application will present a dialog box where the user can input information about the alert, take or attach a picture, and select or create a category to which to assign the alert.

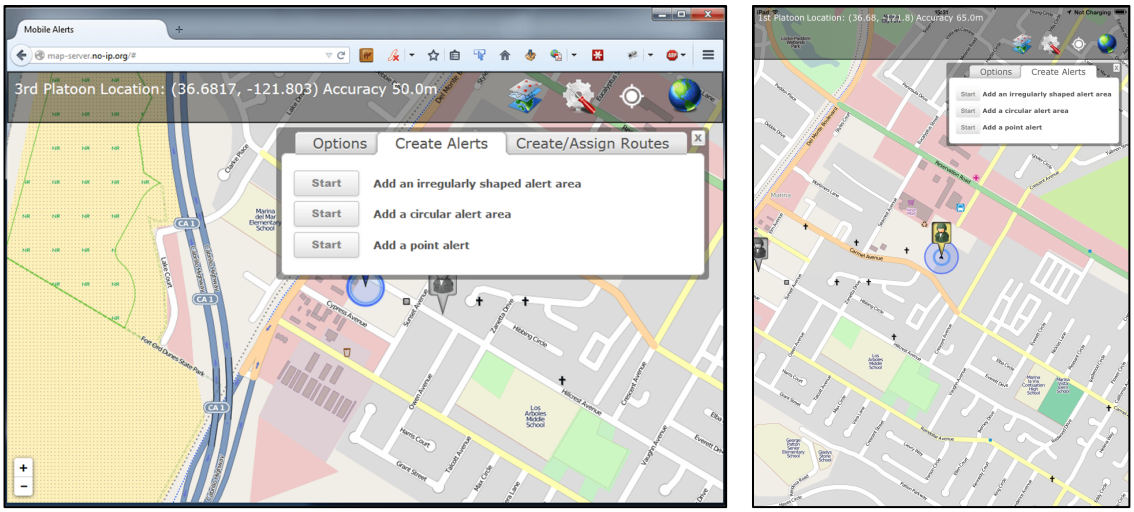

Figure 27. Create Alerts tab, browser (left) and iPad 2 (right)

Once the user selects the type of alert to create, the cursor changes to a crosshair and she can click or touch the map to locate the center of the alert area for a point or circle alert, or create boundary points to define an irregular alert area. Once this has been done, the user is presented with a dialog, as shown in Figure 28, where she can enter information about the alert. If the application is running in a browser the user can upload a picture; on an appropriately equipped mobile device, the user can take a picture. Once the user selects or creates a category to which to add the alert, it is stored in the database and will replicate to all of the other units depending on connectivity.

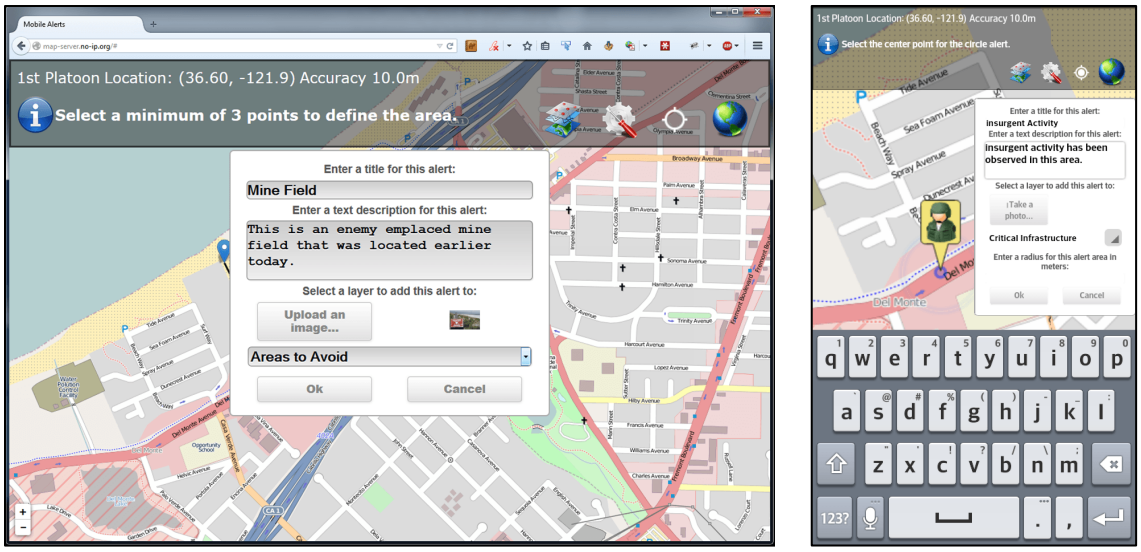

Figure 28. Creating a new alert

c. Create / Assign Routes Tab

The Create / Assign Routes tab, shown in Figure 29, is only available when the application is running in a browser because testing showed that creating accurate routes with touch events on devices with small screens, such as tablets and phones, was extremely difficult. The user has two options for creating a route, automated and manual, as described in Chapter III.

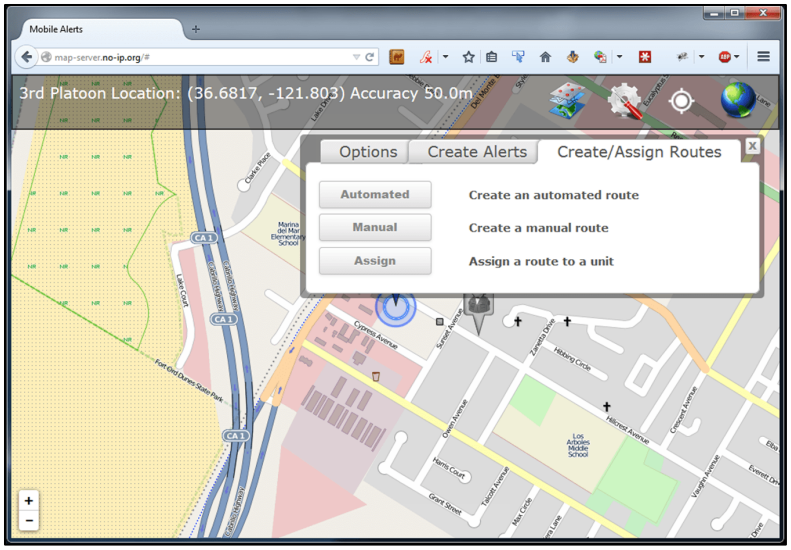

Figure 29. Create / Assign Routes tab

The automated routing option requires network connectivity to the route server but is useful if the user wants the shortest route between two locations, particularly if the locations are far apart. The user is prompted to select two or more waypoints (Figure 30).

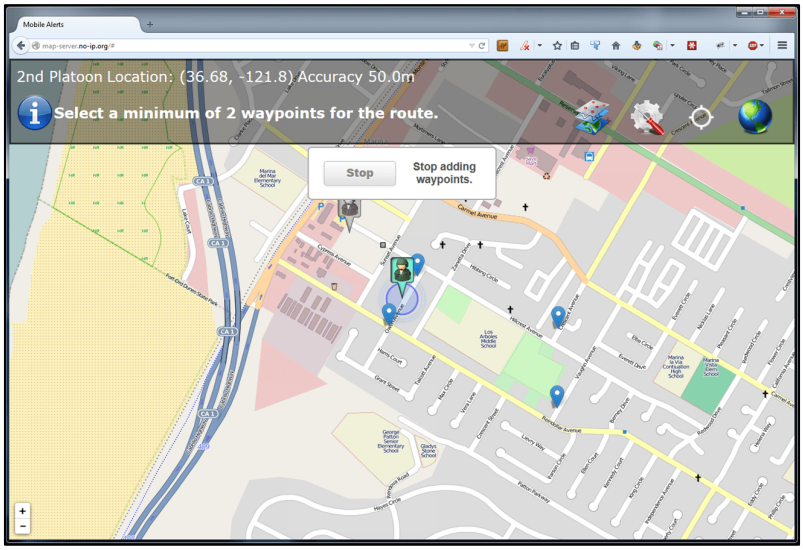

Figure 30. Four waypoints selected

Once the user is done adding the initial waypoints, the application queries the route server and displays one or more proposed routes that includes all of the waypoints, in the order the user selected them, as depicted Figure 31.

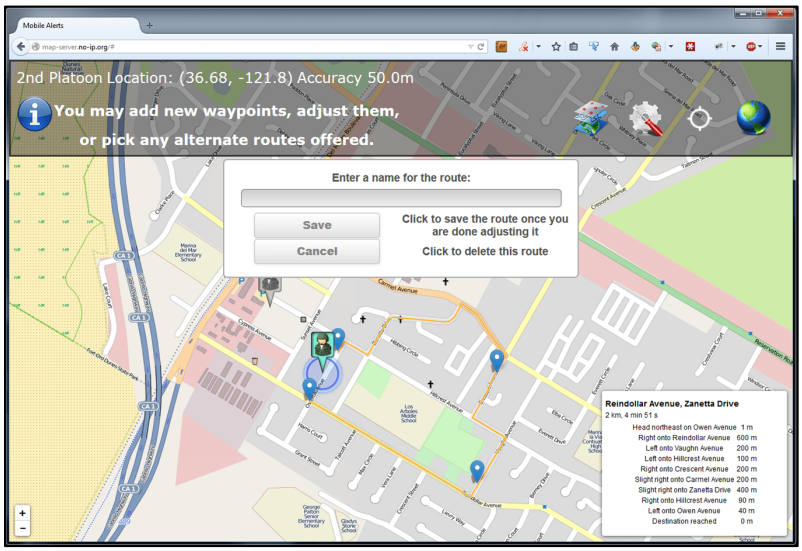

Figure 31. Proposed route between waypoints

The user can continue to adjust the route by dragging the existing waypoints (see Figure 32) or adding additional waypoints (see Figure 33) and the route server will adjust the proposed route as necessary, as shown in Figure 34.

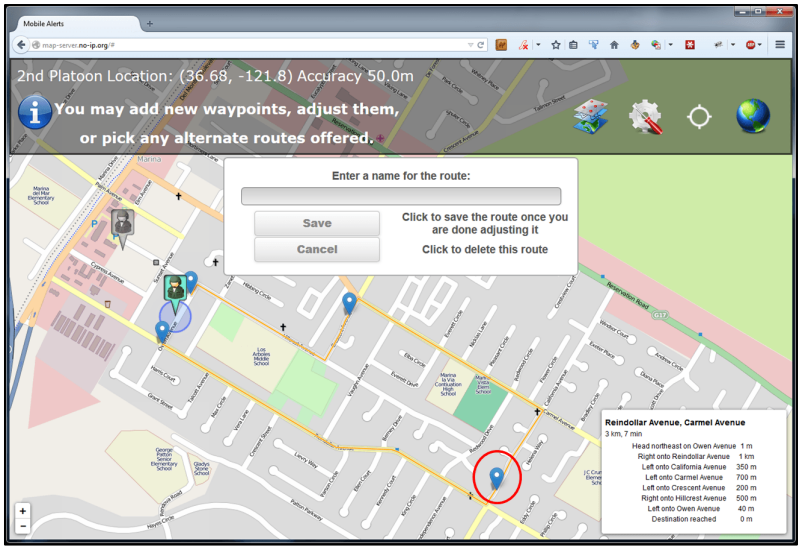

Figure 32. Moving a waypoint (circled)

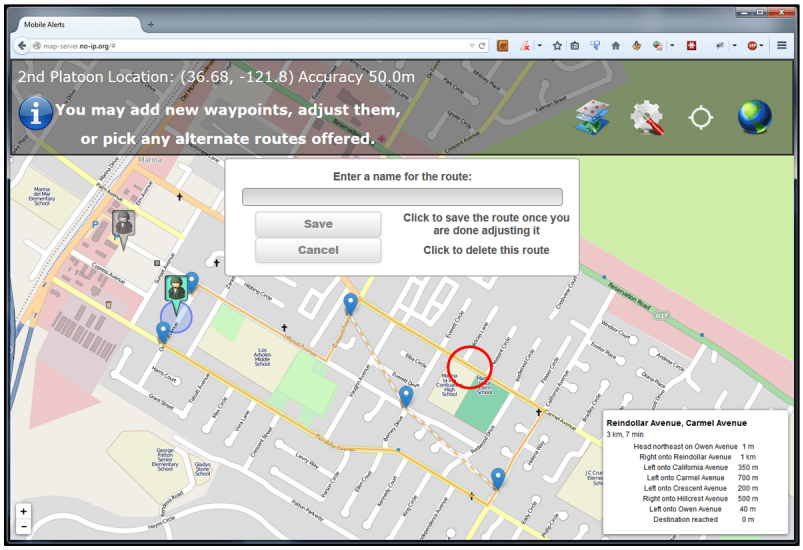

Figure 33. Adding a waypoint by clicking on route and dragging (click/drag point circled)

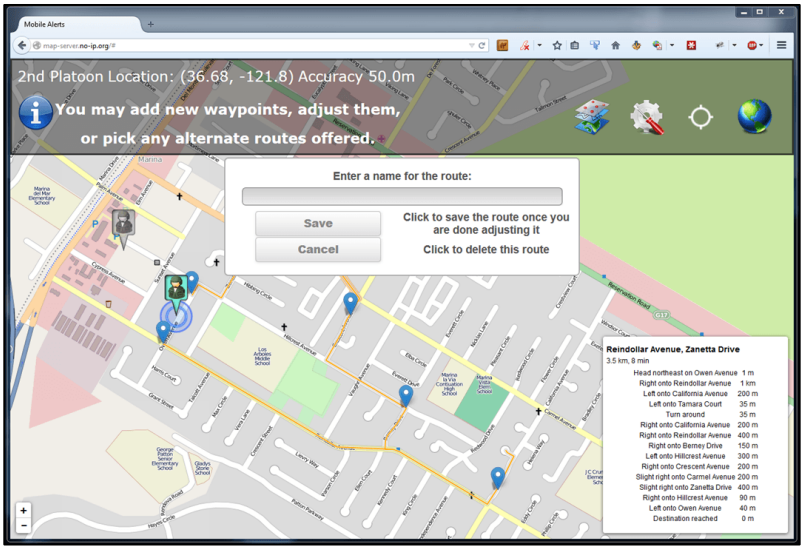

Figure 34. Final route after adjustments

Once the user is satisfied, he enters a unique route name and clicks on the save button and the route is stored to the database and made available to all users.

The Manual Routing option can be used offline, but requires a larger set of waypoints as input for longer routes. The user clicks on the map to generate waypoint icons at each point where the route should turn, and the application draws a route line between them, as shown in Figure 35. Waypoints can be dragged to adjust the route or can be removed by clicking on the respective icon, as depicted in Figure 36. Once the user is satisfied with the route, he can name and store the route as with the Automated option, as shown in Figure 37.

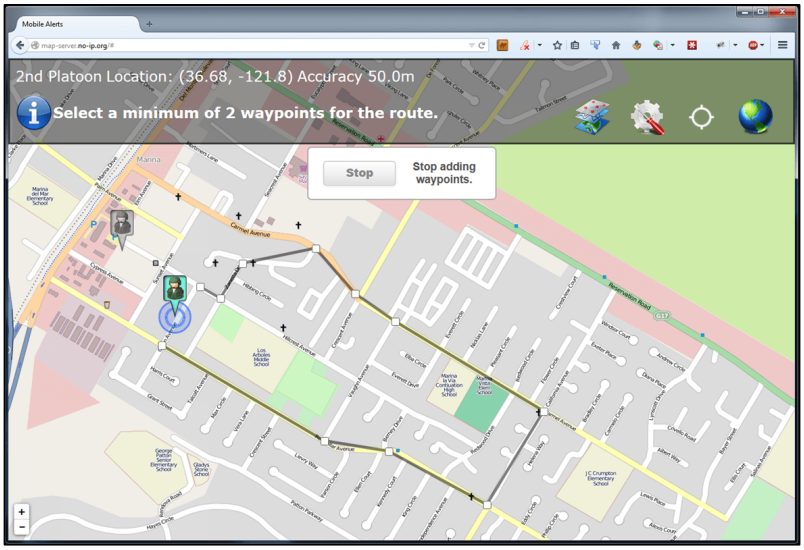

Figure 35. Adding a manual route

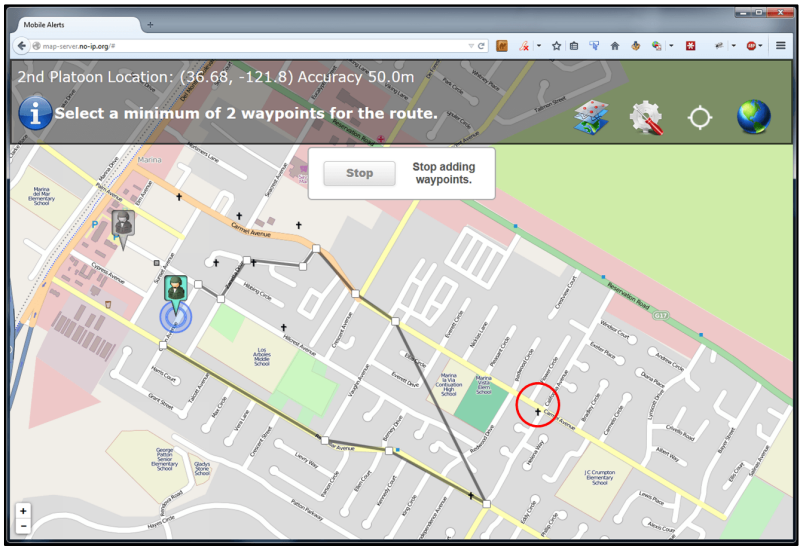

Figure 36. Waypoints adjusted and deleted (circled)

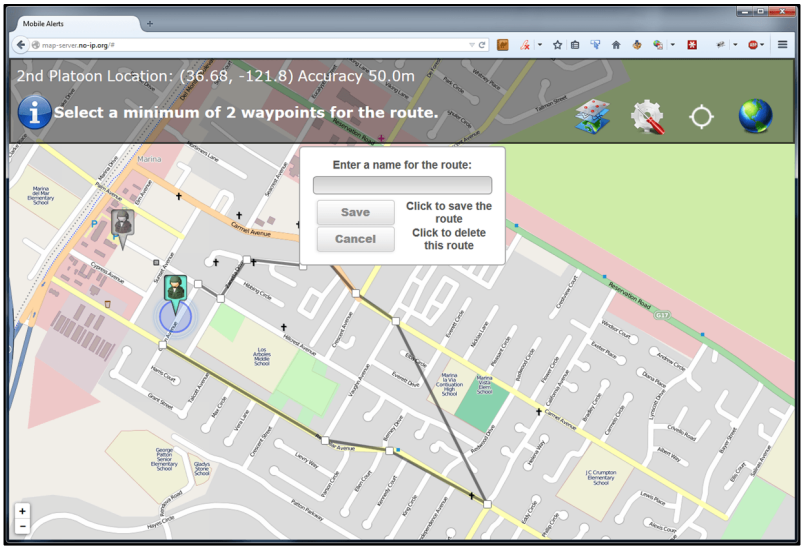

Figure 37. Finalizing a manual route

The Assign route option lets the user assign a route to another unit to follow. This would be used by a command post staff to assign routes to its units and then monitor them as each unit attempts to follow its assigned route. The user selects a route and a unit to which to assign the route using the dropdown menus shown in Figure 38. When the user selects an entry in either the route or unit dropdown, the map zooms to show the route or pans to the unit's location so the user is aware of which route or unit he is selecting.

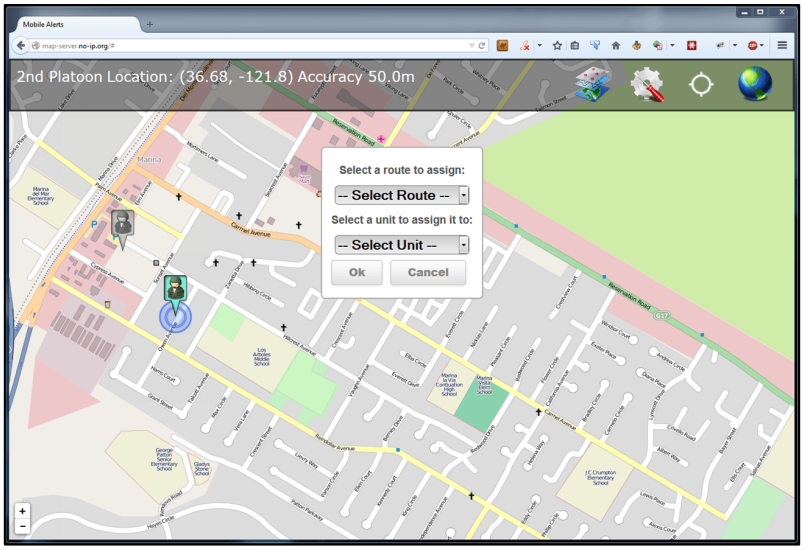

Figure 38. Assigning a route to a unit

4. Alerts

As the user moves around his area of operations, the application continuously tracks his movement and notifies him if he enters an alert area. The application uses audio, visual, and tactile feedback to alert the user. The application vibrates if the device is capable of vibration, plays an audio alert message, and displays a pop-up on the screen in the vicinity of the alert area. The audio alert can be muted by setting the device to silent mode. Examples of the pop-up display are shown in Figures 39 and 40. The user can also display the information pop-up for any alert area by clicking on it. The visual display is dynamically created based on the information provided in the database about that alert. While the information displayed by the prototype system is relatively basic, because the alert pop-up is essentially a web-page contained inside an HTML element, the amount and type of information displayed can easily be changed. In the production application, the pop-up might contain links to more detailed information, a scrollable picture gallery, buttons to bring up a full-screen window of information, etc.

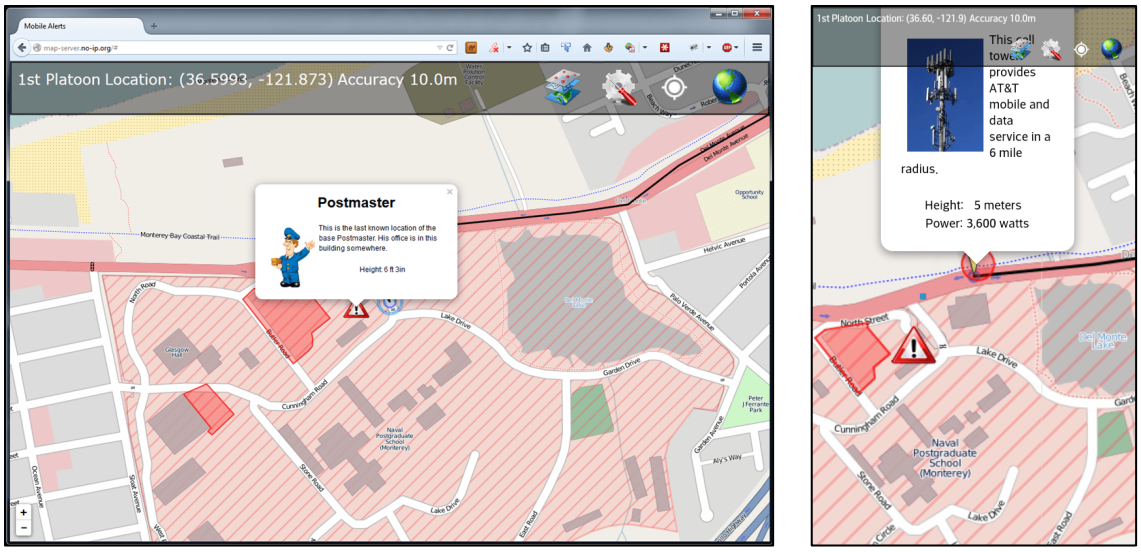

Figure 39. Example alert information display, browser (left) and Android phone (right)

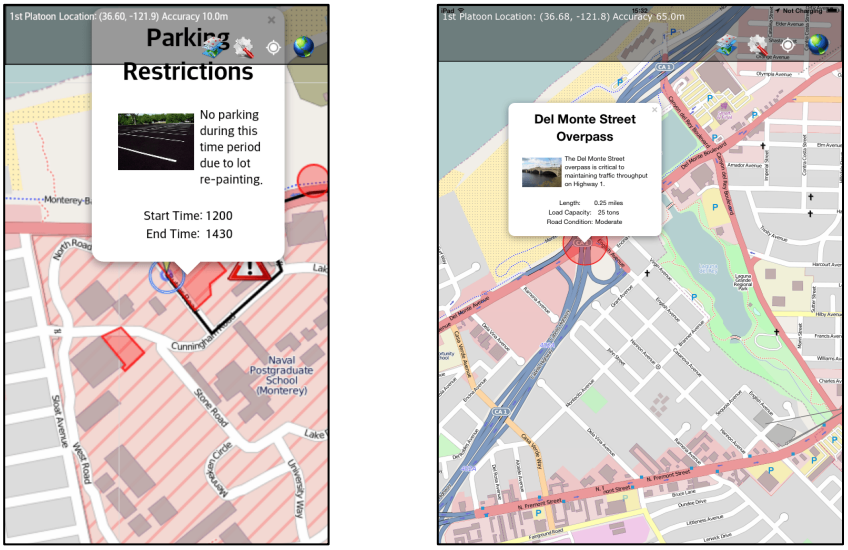

Figure 40. Example alert information display, Android phone (left), iPad 2 (right)

5. Routes

Since connectivity with each unit is not guaranteed, the application initiates a three-way handshake between the assigning unit (Headquarters) and the assigned unit (Unit 1) using the database for communication, as depicted in Figure 41. The three-way handshake ensures that both units are aware of the assignment. While the unit is following the assigned route, both units track the location of the assigned unit. If the assigned unit gets off route by more than 100 meters, plus or minus the GPS accuracy, the application causes the device to vibrate if it is a mobile device and plays an audio alert, displays a message, and turns the route line on the map red to alert the user. The assigning unit's application tracks all units that it has assigned and if one gets off-route, it displays a message and plays an audio message to alert the user of the condition. Once the assigned unit is complete with following the route, that unit's user selects a button on the options menu to notify the assigning unit that he has completed the route. The assigning unit's application then deletes the record from the database.

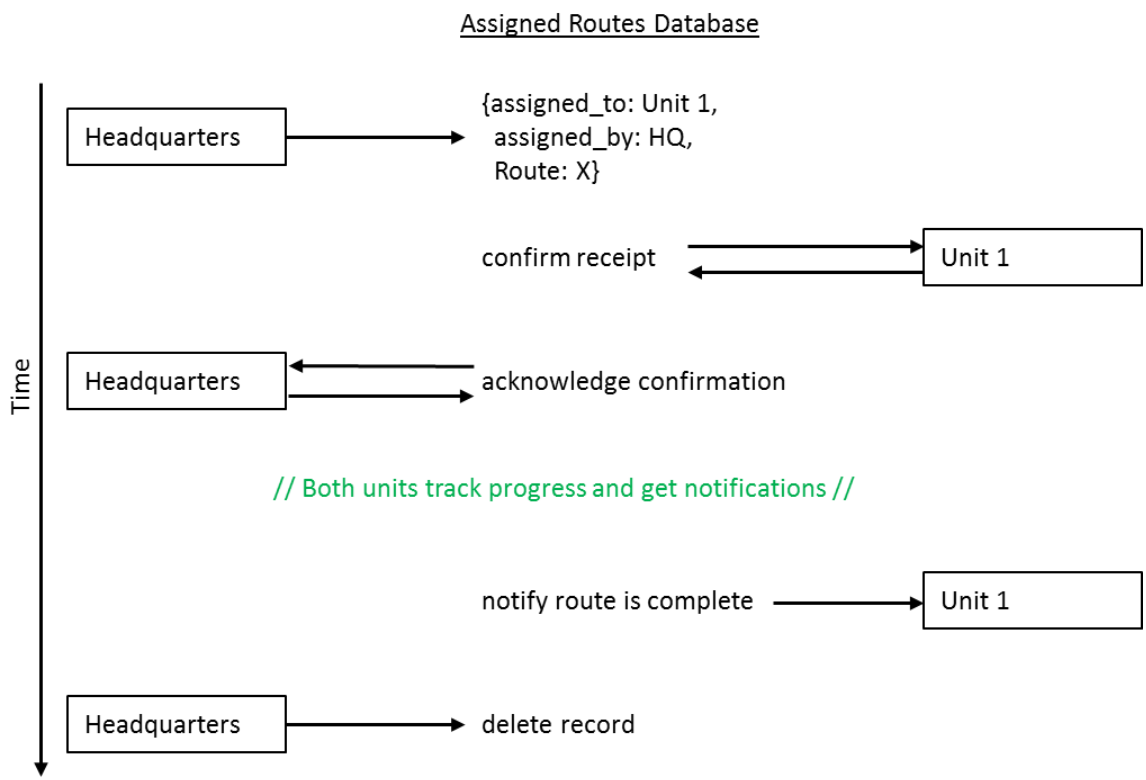

Figure 41. Route assignment process

The application ensures a unit can only be assigned one route by using the assigned_to field as the unique document id for the record. Attempting to assign two routes to the same unit will fail with a database error and the user attempting to assign the duplicate route will be notified that the unit is already assigned a route. Once a unit has confirmed receipt of a route assignment, the unit cannot choose to follow another route or be assigned a new route until it has notified the assigning unit that the route is complete.

Once a unit has been assigned a route and its device has acknowledged that they are following the assigned route, the route is shown on the display using either a green or red line. If the unit is within the accuracy limit, the route line is displayed in green, as shown in Figure 42. If the unit departs from the route, the route line is shown in red showing where the unit is expected to be, as depicted in Figure 43, a visual warning is displayed, the device vibrates (if capable), and an audio alert is played.

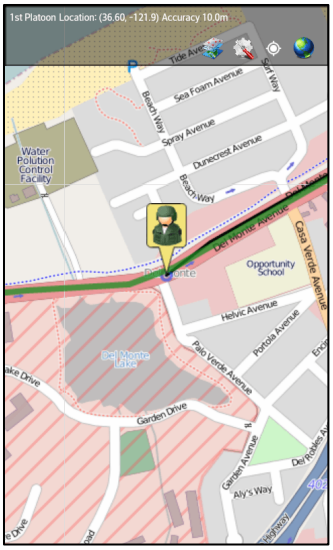

Figure 42. A unit that is on the assigned route

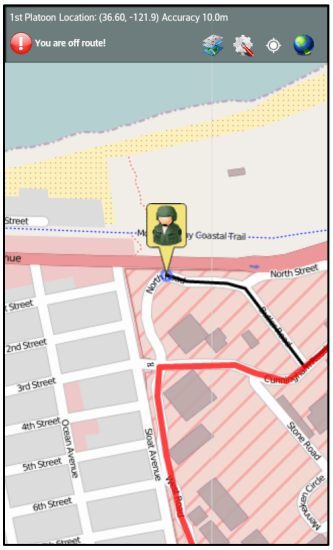

Figure 43. A unit that is off the assigned route

D. TESTING

We tested the application on six different mobile devices and four desktop browsers using several different scenarios to determine whether the application would perform as expected.

1. Devices

The application was tested in four web browsers: Firefox 30.0, Chrome 36.0, Opera 22.0, and Safari 6.1. Since each browser uses a different rendering engine, there are slight differences in the user interface as seen in Figure 44, particularly with implementation of fonts, buttons, and input boxes. The display and layout is similar enough, however, that a user who is familiar with the application on one platform can easily move to another with no additional training.

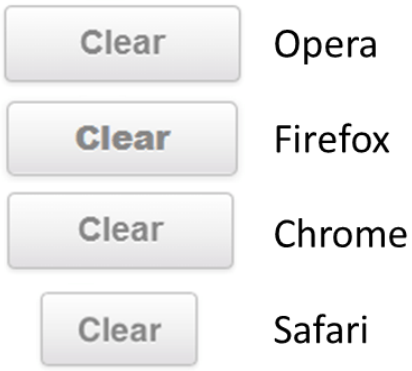

Figure 44. Example of user interface differences: the same button as rendered in four different browsers

In addition to the four web browsers, the application was tested on six mobile devices: an LG Optimus f3 running Android 4.1.2, a Samsung Nexus S running Android 4.1.2, a Samsung Galaxy Tab 10.1 running Android 4.0.4, an iPad Generation 1 running iOS 5.0, an iPad 2 running iOS 7.1.1, and an iPhone 4 running iOS 7.1.1. The user interface on all of the devices showed similar differences to those experienced with the browsers. In addition, due to the differences in keyboard and user input processes for the different mobile browsers and operating systems, the user input for dropdowns and keyboard entry on the mobile application does appear different on each platform, even within the same operating system, as shown in Figures 45 and 46.

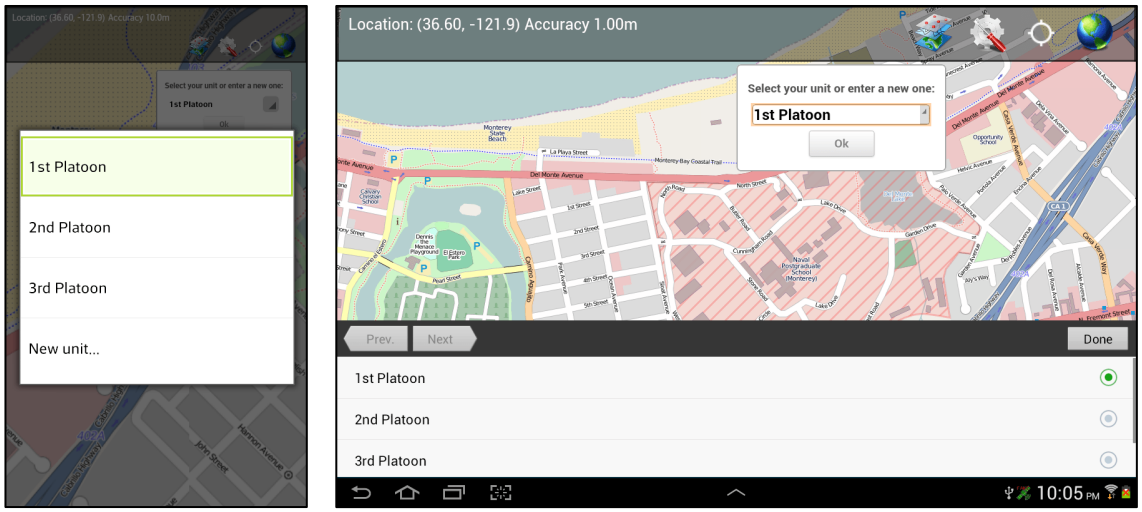

Figure 45. Dropdown input differences, Android phone (left) and Android tablet (right)

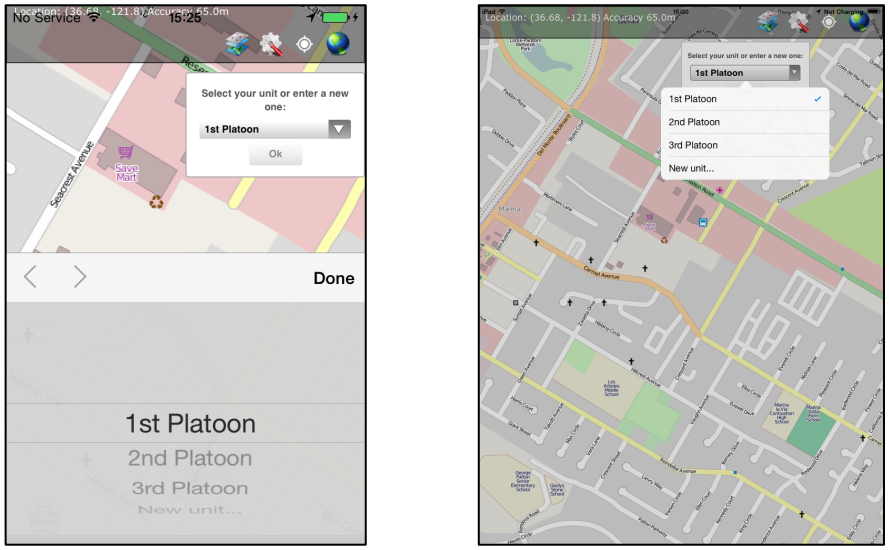

Figure 46. Dropdown input differences, iPhone (left), iPad 2 (right)

2. Scenarios

We tested two scenarios for using the mobile application. Usability testing was focused on determining how well the user interface worked when the

application had full Internet connectivity. Cache testing focused on how well the map tile cache functioned.

a. Usability Testing

The application was tested using a 3G mobile data connection on the LG Optimus f3 using both walking and driving scenarios to determine whether it would accurately track an assigned route and provide situational awareness alerts to a user when entering alert areas along the route. The phone was assigned a route from another unit using the application in a desktop browser. The browser session was then recorded while the phone was transported along the assigned route. The application successfully tracked the location of the phone both while driving and walking. Experimentation determined that five seconds worked as an appropriate GPS update interval, and when connectivity was available, database updates every 10 seconds provided a reasonable tradeoff between timeliness and bandwidth usage. The session recorded from the desktop browser accurately recorded the position of the phone along the route with minimal delay. The phones successfully notified the user when a GPS update was received while inside the test alert areas and when he deviated from the route. When the mobile device loses connectivity, the application still continues to track correctly and when connectivity is re-established, the unit icon being shown on other devices for that phone does move to the correct current position.

b. Cache

The offline cache function was tested for usability with several desktop browsers and on mobile devices including two Android phones, an Android tablet, and an iPad II. Detailed test data to determine actual download sizes and speeds were compiled using the Firefox and Chrome browsers with Gigabit Ethernet, and on an LG Optimus f3 with 3G service and Wi-Fi. Each download

test produced a waterfall chart as seen in Figure 47 that was saved from the browser in HTTP Archive format and then exported to Microsoft Excel for analysis.

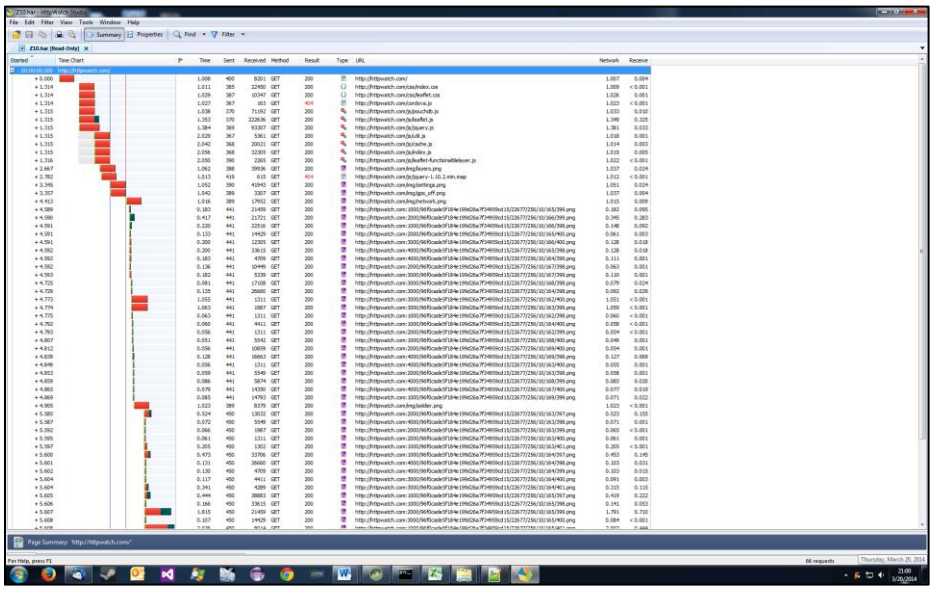

Figure 47. Waterfall chart of download times

Actual testing was completed for zoom levels 10 through 14, as shown in Table 3, and the times shown reflect the results averaged over three tests. Testing with more than five zoom levels caused Chrome and Firefox to crash, and more than four levels caused the mobile browser on the LG Optimus to crash. Our hypothesis is that this is due to the number of queued download requests exceeding the size of the browser's pending download queue since the application asynchronously feeds the download requests to the browser as quickly as it can create them. For the sixth level this would result in 16,641 pending requests being added to the queue, likely while the browser is still processing the first several hundred downloads. This problem could be mitigated by buffering the requests and limiting the maximum number of requests sent to the browser at any given time.

	Zoom Level	10	10 - 11	10 - 12	10 - 13	10 – 14
	Tiles Required	25	106	395	1484	5709
	Predicted Download Size (including headers)	275.00 Kb	1.17 Mb	4.35 Mb	16.32 Mb	62.80 Mb
	Actual Download Size (including headers)	344.26 Kb	1.09 Mb	4.09 Mb	16.43 Mb	52.62 Mb
Download Times in Seconds						
3G (384 Kbps) LG Optimus f3	Calculated	5.7291667	24.29167	90.52083	340.0833	1308.3125
	Actual	9.42	25.824	104.371	355.002	
802.11n (600 Mbps) LG Optimus f3	Calculated	0.0036667	0.015547	0.057933	0.217653	0.83732
	Actual	3.364	12.943	59.09	368.249	
802.3 Gigabit Ethernet	Calculated	0.0022	0.009328	0.03476	0.130592	0.502392
	Actual	3.269	15.033	52.863	364.566	2784.359

Table 3. Actual download sizes and times for different connection types

E. SUMMARY

This chapter has provided a detailed breakdown of the functions the application performs and how they were implemented. It also explained the testing that was performed to determine how well the application would track a user along a route and provide situational awareness alerts. The next chapter outlines the conclusions we reached after developing and testing the application and discusses additional enhancements that could be made in the future.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSIONS AND FUTURE WORK

Our research led us to develop several conclusions about the feasibility of using a single-baseline mobile application to improve situational awareness. Our research also identified a number of areas for future work that would improve the prototype.

A. CONCLUSIONS

We developed a single-baseline mobile application that runs on multiple types of devices, tracks a user's location, and improves situational awareness by alerting the user to conditions along a patrol route. Our prototype application demonstrates that it is possible to use Apache Cordova and HTML5, CSS3, and JavaScript to produce an application that runs on a variety of devices and operating systems without maintaining operating system specific code. In addition, our prototype application demonstrates a method for taking multiple types of geolocated data from different notional database systems, generate appropriate alerts and present to the users information based on their proximity to relevant data elements. While the test information shown in the prototype application was relatively simple, it does show that it is possible to improve a patrol's situational awareness by providing visual, auditory, and tactile alerts when the patrol enters an area where the database contains information that might be relevant to the patrol.

Our prototype application demonstrates some of the advantages of a web-based, single source code application, namely that the application can be made to look and function identically on different devices. Any user familiar with the application will be able to use it on any compatible device without a need for additional training. Since the application is not constrained to a single type of device or operating system, it should work well in a situation where different government agencies are cooperating on an operation but where each uses a

different type of mobile device. Code maintenance and baseline tracking is simplified and adding or modifying a feature makes the necessary changes suitable for all supported devices.

Our prototype application highlights some of the challenges associated with developing a web-based mobile application including operating system differences and browser limitations. While Cordova does minimize the differences in code required for different devices and operating systems, considerations still have to be made for some of the device dependencies. One example is that some of the mobile operating systems consider file names to be case sensitive while others don not; this resulted in errors when testing with several of the devices. Another example is the differences in user input functionality, as highlighted in Chapter IV, particularly the soft keyboard, dropdown menus, and text entry. These user interface differences might be mitigated by developing pure HTML/CSS3/JavaScript replacements (Satterfield & Garrison, 2014) at the expense of additional code complexity, application load time, and memory usage. Additionally, HTML5 implementation is not complete or standard across the different mobile browsers. Many of the newer HTML5 features like WebRTC, the Audio element and associated codecs, and the FileSystem API are not available on all devices without using a Cordova plugin that implements the functionality in native code.

The Cordova plugins rely on native code that is specific to each operating system, so while a specific function is accessed the same way on every device, the results may vary on some of the devices. For example, the vibrate function allows the application to specify a duration for the vibration. Android, Blackberry, and Windows Phone use the specified duration; iOS ignores the requested duration and vibrates for two seconds (Apache Software Foundation, 2014). In addition to the minor differences in operation, standard Cordova plugins are not available for some of the more advanced features available on mobile devices such as Bluetooth or Near Field Communication.

While our goal was minimizing the reliance of the application on external services in order to maximize the application's capability to operate offline, we determined that including full-featured map display, route tracking, and data sharing all required external services. Our review of current mobile device applications and code libraries revealed that while there is interest in providing some of these functions in an offline setting, much of the commercial development in the map display, routing, and data sharing assumes that the mobile device has a persistent Internet connection.

B. FUTURE WORK

Our prototype application was designed to show how a single baseline web-based application could improve the situational awareness of a patrol by providing alerts to relevant information along a patrol route. There are a number of areas where additional research could be performed to improve the capabilities of our application. These areas include user login, identification, and authentication; external database integration; improved offline functionality; improved map caching; and dynamic HTML generation from database objects.

1. User Login, Identification, and Authentication

Our prototype application identifies its user account based on the unit selected by the user when the application loads. This unit is required to be uniquely identified by name; the database will reject the creation of two units with the same name. The application does not currently have a method to validate the user's selection of his unit or to tie that unit to a unique device. If multiple users select the same existing unit name upon application load, this will cause confusion because both devices will report their position as that unit. Other instances of the application would see that unit's icon bounce rapidly back and forth between the two reported positions. We investigated methods for uniquely tying an application instance and unit to a device using the IMEI or MAC address of the device, or storing a unique id on the device's file system on the application's first run, however neither method proved feasible. A method needs

to be determined to authenticate users and associate a unique user to a specific device. Such authentication is essential when distributing sensitive command and control (C2) or control and incident response (CIR) data to ensure only authorized personnel are granted access. Further, confidentiality of data should also be addressed, perhaps by incorporating suitable encryption of data at rest and in transit methodologies according to pertinent organizational standard operating procedures.

2. External Database Integration

One of the goals of this thesis was to determine how to alert a patrol based on information from a variety of different databases. One of the reasons we selected a NoSQL database is that each record can have a unique set of fields that enables the application's database to store and display information taken from a variety of sources. The U.S. military and other law enforcement agencies have a vast number of databases, of varying classifications or confidentiality levels that contain data that would be relevant to a patrol. Having each mobile instance of the application attempt to access each of these databases to query for relevant information would require significant bandwidth, so our system architecture was designed with a central database that would aggregate the information from external databases, put it into a standard format, and then replicate it to the mobile instances when connectivity is available. This aspect of the system architecture should be evaluated more extensively to determine if it is the best way to accomplish this goal, particularly in light of the possible security implications of aggregating the information from multiple databases in one place. In addition, we did not develop the process that would be required to query the external databases for relevant information; this process should be explored in detail and tested.

3. Offline Functionality

Our prototype application has significant limitations when there is no network connection. We assume that the application will have network

connectivity when it is initialized so that it can synchronize its databases and cache the map tiles. Once the application loses its connection it will continue to operate based on the data it has stored internally, but as a situational awareness tool, the longer it has no network connection the less relevant its data becomes. One of our goals that was not realized is the ability to share new data directly between two instances of the mobile application. There is no Cordova plugin that supports Bluetooth or WiFi Direct transfers between two mobile devices. The development of a Bluetooth or WiFi Direct plugin for Cordova would enable two units within wireless range to synchronize their databases directly without the requirement for cellular or wireless infrastructure.

4. Map Cache

The algorithm used to cache the map tiles is simple and effective, but it is not efficient or user friendly. The prototype application caches all of the map tiles within approximately 20 kilometers of the user's current location. If the user moves outside of that area, they have to tell the application to re-fill the cache based on the new current location, which requires the device to be online. The old map tiles are not discarded in this process. If the device storage fills up, the user can select to clear the entire cache and start over. In addition, the map tiles are stored as uncompressed PNG files on the mobile device or as blobs in a local database in the browser. If two identical tiles appear in separate locations on the map, ocean tiles for example, both copies will be stored. Additional research into automating the caching of tiles, storing them more efficiently, using virtual links to common tiles, or dynamically rendering the tiles on the device should be explored. Consideration should also be given to digitally signing tiles to ensure tile content integrity.

5. Dynamic HTML Generation for Alerts

The information provided in the alerts is intended to provide details based on what information is available about that alert and to provide a capability for more detailed drill-down. The prototype application dynamically creates alert

window based on the data available in the database object that generated the specific alert, but it is currently limited to only a few fields. For example, the function that creates the alert window checks for an associated photo attachment and displays it if one is available; it will also format and display a list of characteristics if such a list is present. A more full-featured version of the application should be able to analyze the JSON object returned by the database and dynamically create an alert window that is appropriately sized for the device with opportunities for drill-down by the user. For example, if the database object contains multiple photo attachments, the alert popup might provide a link by which the user can access a photo gallery to scroll through all of the photos. It could also provide a way for the user to open the alert information in a full-screen window, include additional text fields, and allow the user to scroll through the information.

The goal for this research was to develop an architecture and prototype system to demonstrate that a handheld assistant could enhance the effectiveness and security of patrol units while expediting the planning process. Our prototype successfully demonstrated that a hardware-independent mobile application could be developed that will track the user's location and provide alerts when the user is in the vicinity of geospatially tagged information. The future of this research lies in improving the functionality that was implemented in the prototype, expanding the network communication abilities to include Bluetooth and WiFi Direct, addressing the security requirements, and developing the ability to pull relevant data from the multitude of existing DOD databases.

SUPPLEMENTARY MATERIAL

The application code for the mobile application has been included as supplementary material for this thesis. This material includes the contents of the "www" folder structure as well as all HTML, JavaScript, CSS, sound, and icon files necessary to compile the mobile application using Cordova. Also included is the JavaScript file for the NodeJS proxy server that was used to enable the mobile application to connect with the various servers described in Chapters III and IV.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

Adobe Systems Inc. (2011). October 3. Adobe announces agreement to acquire Nitobi, creator of PhoneGap. Adobe Systems Inc. Press release. Retrieved from http://www.adobe.com/aboutadobe/pressroom/pressreleases/201110/AdobeAcquiresNitobi.html

Agafonkin, V. (2014). Leaflet (version 0.7.3) [software]. Available from http://leafletjs.com/download.html

Allensworth, G. (2014). MobileMapStarter [software]. Available from https://github.com/gregallensworth/MobileMapStarter

Anderson, J. C., Lehnardt, J. & Slater, N. (2009). *CouchDB: The definitive guide* (1st ed.). Sebastopol, CA: O'Reilly Media, Inc.

Android Open Source Project. (2014a). Android NDK. https://developer.android.com/tools/sdk/ndk/index.html

Android Open Source Project. (2014b). The Android source code. Retrieved from http://source.android.com/source/index.html

Apache Software Foundation. (2014). Apache Cordova documentation. Retrieved from http://cordova.apache.org/docs/en/3.3.0/index.html

Apple Inc. (2014a). App Store review guidelines. Retrieved from https://developer.apple.com/appstore/resources/approval/guidelines.html

Apple Inc. (2013). *Start developing iOS Apps today*. Cupertino, CA: Apple Inc.

Apple Inc. (2014b). Which developer program is for you? Retrieved from https://developer.apple.com/programs/which-program/

Bu, G. (2011). Entire network icon [image]. Retrieved from http://kidaubis.deviantart.com/art/Reality-104372241.

Bureau of Justice Assistance. (2012). Reducing crime through intelligence-led policing. Retrieved from https://www.ncirc.gov/documents/public/reducing_crime_through_ilp.pdf

Chamberlain, R. (1996). Calculating distances on the surface of the earth. Retrieved from http://www.faqs.org/faqs/geography/infosystems-faq/

Coelho, E. (2007). Apps service manager icon [image]. Retrieved from http://www.softicons.com/system-icons/crystal-project-icons-by-everaldo-coelho

Dees, I. & Weait, R. (2013). Manually building a tile server. Retrieved from http://switch2osm.org/serving-tiles/manually-building-a-tile-server-12-04/

Defense Information Systems Agency. (2014). GCCS-joint. Retrieved from http://www.disa.mil/Services/Command-and-Control/GCCS-J.

Fitts, P. (1992). The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology: Genera*l, *121*(3): 262.

Flynn, M., Pottinger, M., Batchelor, P. (2010). *Fixing intel: A blueprint for making intelligence Relevant in Afghanistan*. Washington, DC: Center for a New American Security.

Forlines, C., Wigdor, D. Shen, C. & Balakrishnan, R. (2007). Direct-touch vs. mouse input for tabletop displays. *CHI Conference Proceedings*, *1*: 647–656.

Frain, B. (2012). *Responsive Web Design with HTML5 and CSS*3. Birmingham, UK: Packt Publishing.

Fuentes, R. & Hunt, J. (2006). Operation LEAD: New Jersey's statewide response to Louisiana in the aftermath of Hurricane Katrina. *Police Chief*, *73*(2): 36–53.

García, R., de Castro, J.P., Verdú, E., Verdú, M.J. & Regueras, L.M. (2012). Web map tile services for spatial data infrastructures: Management and optimization. In C. Bateira (Ed.), *Cartography: A tool for spatial analysis*. Available from http://www.intechopen.com/books/cartography-a-tool-for-spatial-analysis/web-map-tile-services-for-spatial-data-infrastructures-management-and-optimization

General Dynamics Inc. (2012). *Tactical Ground Reporting (TIGR) System* [datasheet]. Arlington, VA: General Dynamics. Retrieved from http://www.gdc4s.com/Documents/Programs/TIGR Handout-Final.pdf

Google Inc. (2014). Google Maps API: Map types. Retrieved from https://developers.google.com/maps/documentation/javascript/maptypes

Harvey, D. & Lawson, N. (2014). PouchDB. Retrieved from http://pouchdb.com/

Icons-Land. (2014). Layers Icon [image]. Retrieved from http://www.icons-land.com

Koch, P. (2014). Touch Table. Retrieved from http://www.quirksmode.org/mobile/tableTouch.html

Lacey, M. (2010, January 20). US troops patrol Haiti, filling a void. *The New York Times*. A1.

Leroux, B. (2012). PhoneGap, Cordova, and what's in a name? Retrieved from http://phonegap.com/2012/03/19/phonegap-cordova-and-what%E2%80%99s-in-a-name/

Lie, H. W. & Bos, B. (2005). *Cascading style sheets: Designing for the web*. Upper Saddle River, NJ: Addison Wesley.

Liedman, P. (2014). Leaflet Routing Machine [software]. Retrieved from https://github.com/perliedman/leaflet-routing-machine

Luo, T., Hao, H., Du, W., Wang, Y., & Yin, H. (2011, December). Attacks on WebView in the Android system. *Proceedings of the 27th Annual Computer Security Applications Conference*: 343–352. ACM.

Luxen, D. (2014). *Open source routing machine wiki*. Retrieved from https://github.com/DennisOSRM/Project-OSRM/wiki

Mahdavi, N. (2014). Leaflet plotter [software]. Available from https://github.com/scripter-co/leaflet-plotter

Mapbox Inc. (2014). Mapbox: an open platform. Retrieved from https://www.mapbox.com/foundations/an-open-platform/

Microsoft Corporation. (1999). *FAT: General overview of on-disk format* [white paper]. Redmond, WA: author.

Mozilla Foundation. (2014). Firefox OS. Retrieved from https://developer.mozilla.org/en-US/Firefox_OS

National Oceanic and Atmospheric Administration & National Aeronautics and Space Administration. 2007. Seasonal blue marble [image]. Retrieved from ftp://public.sos.noaa.gov/land/blue_marble/seasonal_blue_marble/

Naval Surface Warfare Center. 2013*. Software version description (SVD) for Joint Battlespace Viewer (JBV*). Panama City, FL: United States Navy.

OSM map on Garmin/IMG file format. (2013). Retrieved July 6, 2014 from OpenStreetMap Wiki: http://wiki.openstreetmap.org/w/index.php?title=OSM_Map_On_Garmin/IMG_File_Format&oldid=952578

Paterson, R., Greenberg, J. & Green, H. (2010). Command post of the future: Successful transition of a science and technology initiative to a program of record. *Defense AR Journa*l 17, *1*(53): 3–26.

Pennington, J.V. (2008)*. COIN patrolling: Tactics, techniques, and procedure*s. Fort Leavenworth, KS: Center for Army Lessons Learned, Combined Arms Center.

Plotz, M. (2013). Getting jQuery Mobile 1.4 and PhoneGap 3.1 to work together. Retrieved from http://www.devx.com/wireless/getting-jquery-mobile-1.4-and-phonegap-3.1-to-work-together.html

Rendering. (2014). *OpenStreetMap wiki*. Retrieved July 8, 2014 from http://wiki.openstreetmap.org/wiki/Rendering

Rivera, J. & van der Meulen, R. (2013, November 14). Gartner says smartphone sales accounted for 55 percent of overall mobile phone sales in third quarter of 2013. Press release [Gartner]. Retrieved from http://www.gartner.com/newsroom/id/2623415

Sample, J. & Ioup, E. (2010)*. Tile-based geospatial information systems: Principles and practice*s. New York: Springer.

Satterfield, J. & Garrison, R. (2013). Virtual keyboard. Retrieved from http://mottie.github.io/Keyboard/index.html

Sicking, J. (2012, July 5).Why no FileSystem API in Firefox? Retrieved from https://hacks.mozilla.org/2012/07/why-no-filesystem-api-in-firefox/

Slippy map. (2014). *OpenStreetMap wiki*. Retrieved July 8, 2014 from http://wiki.openstreetmap.org/wiki/Slippy_Map

Smyrnow, I. (2014). Leaflet FunctionalTileLayer [software]. Available from https://github.com/ismyrnow/Leaflet.functionaltilelayer

Stöckli, R., Vermote, E., Saleous, N., Simmon, R. & Herring, D. (2005)*. The blue marble next generation: A true color earth dataset including seasonal dynamics from MODIS*. Greenbelt, MD: NASA Earth Observatory.

Tile disk usage. (2013). Retrieved July 14, 2014 from OpenStreetMap Wiki: http://wiki.openstreetmap.org/wiki/Tile_disk_usage

United States Army. (2008)*. Training for full spectrum operation*s. Washington, DC: author.

United States Marine Corps. (2000)*. Scouting and patrolling*. Washington, DC: author.

United States Marine Corps. (2003). *Intelligence operations*. Washington, DC: author.

Vetter, C. (2010). *Fast and exact mobile navigation with Openstreetmap data*. Master's thesis, Karlsruhe Institute of Technology.

Willis, D. (2013). Bring your own device: the facts and the future. Stanford, CT: Gartner Inc.

Wilson, C. & Kinlan, P. (2013, March 13). Touch and mouse: together again for the first time. Retrieved from http://www.html5rocks.com/en/mobile/touchandmouse/

Yug & Cfaerber. (2006). Bitmap VS SVG [Digital image]. *Wikimedia Commons*. Retrieved from http://commons.wikimedia.org/wiki/File:Bitmap_VS_SVG

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California